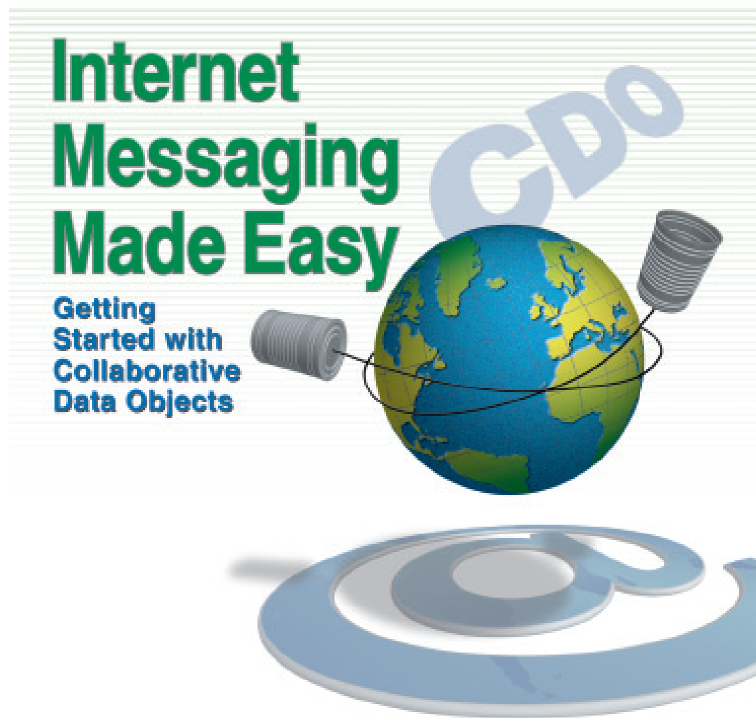
October 2000, Volume 6, Number 10



Cover Art By: *Arthur Dugoni*

ON THE COVER

7 On the 'Net

Internet Messaging Made Easy — Kristen Riley

Need to interact with Microsoft Exchange/Outlook? Ms Riley introduces the main CDO library with example applications that gather address lists, retrieve and send e-mail, and more.

FEATURES

12 Dynamic Delphi

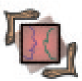
Automating Word: Part II — Ron Gray

Mr Gray completes his two-part series. This month the focus turns to the Word components available in Delphi 5, and how to link and embed documents using OLE.

17 OP Tech

Database Persistent Objects: Part II — Keith Wood

Last month, he showed us classes that can automatically store their published properties in a relational database. This month, Mr Wood shares a form wizard for creating them easily.



21 On Language

A Quick Way to Shortcuts — Bill Todd

Mr Todd examines Windows shortcuts in detail, then shows us how to build a custom component you can use to create and modify shortcuts in any folder.

26 Columns & Rows

A Practical Guide to ADO Extensions: Part I

— Alex Fedorov and Natalia Elmanova

Mr Fedorov and Ms Elmanova demonstrate the use of two ADO extensions from Delphi: ADO Extensions for DDL and Security, and the Jet and Replication Objects library.

REVIEWS

32 IBObjects 3.4

Product Review by Robert Leahey

35 *Delphi Graphics and Game Programming Exposed! with DirectX*

Book Review by Alan C. Moore, Ph.D.

DEPARTMENTS

2 **Symposium** by Jerry Coffey

3 **Delphi Tools**

6 **Newsline**

36 **Best Practices** by Clay Shannon

38 **File | New** by Alan C. Moore, Ph.D.

Symposium

Last Man Standing

The mood at the 11th Annual Borland Conference (held this July in San Diego) was upbeat. This hasn't always been the case; there were several years where many thought they were attending the last Borland conference. In particular, the Orlando conference in 1994 had the feel of a final fond gathering of friends before dissolution. Delphi came out the next year. Which reminds me... this year, some of the Delphi R&D team were wearing baseball caps with the snappy slogan: "Delphi: Saving Borland One Quarter at a Time." Gotta love that.

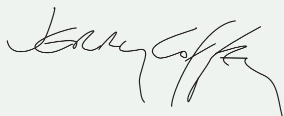
And the Borland community should be jubilant. There's plenty to be happy about: Delphi 6, Kylix, and — lo and behold — JBuilder for the Macintosh. I didn't think I could be excited by a Macintosh product, but seeing JBuilder fire up on an iMac was a genuine thrill. Perhaps I could create something for my son at school... For the first time in a long time, Borland has direction, new markets to conquer, and every reason to expect success in those markets.

There was also a novel aspect to this year's conference: the Kylix previews. And, as exciting as it is, I don't mean Kylix *per se*. Rarely has a product been shown to a conference crowd this early in its development cycle. Since the Kylix IDE wasn't yet functional, the speakers were using Emacs as the editor, then creating the executables with a command-line compiler. Delphi was never shown in public that early.

So what has changed? How can Borland reveal Kylix at such an early stage? There's literally no competition. When Delphi was being developed, Borland had to keep it from prying eyes at Microsoft, Symantec, and elsewhere. There's now only one competitor left, Microsoft, and they're not about to create a RAD tool for Linux. Now that we live in the Pax Microsofta, there's no need to keep the kimono closed. In retrospect, the Bruce Willis vehicle *Last Man Standing* might have been a better choice than *The Matrix* as the conference motif.

Perhaps the most exciting event occurred after the conference, when I saw the August 2000 issue of *Visual Basic Programmer's Journal*. *VBPJ* is the premiere VB magazine, and this month it came in a polybag containing a fully-functional, 60-day evaluation version of Delphi 5 Enterprise. There's also a full-page ad inside pointing readers to www.vbforlinux.com. This is the best news to come out of Borland in recent memory. And I don't just mean the polybag coup. What's important is that Borland thinks it can take on Microsoft directly in the RAD marketplace. I think they can as well — always have — but it's immensely heartening to see Borland marketing Delphi so aggressively.

Thanks for reading.



Jerry Coffey, Editor-in-Chief
jcoffey@informant.com

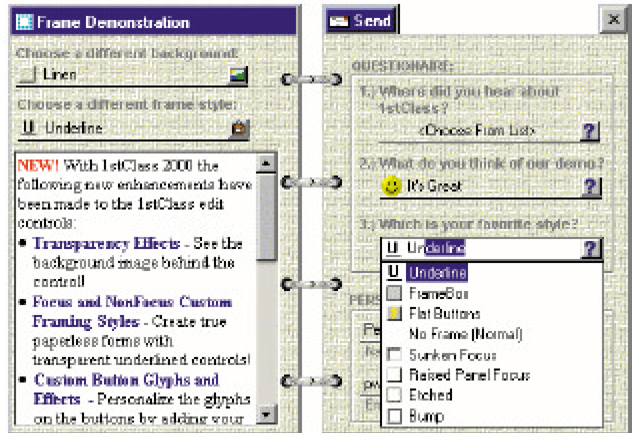
Woll2Woll Announces 1stClass 2000

Woll2Woll Software announced the availability of *1stClass 2000*, a new upgrade to its visual component suite for Delphi and C++Builder.

1stClass 2000 offers image-shaped forms and buttons, data-aware and non-data-aware treeview controls, an Office 97 Outlook Bar Style container control, the Imager control, and a statusbar with many built-in styles and the ability to size panels proportionally. You can use edit controls like the font, color, tree, and image combos that can be embedded directly in an InfoPower Grid.

1stClass 2000 offers several new features, including support for custom framing and transparency effects; support for the same button and glyph effects that can be found in the InfoPower 2000 VCL; improved image painting in 256-color environments when using the following controls: TfcImageBtn, TfcImager, TfcImageForm; and C++Builder 5 compatibility (Professional version).

Woll2Woll Software
Price: Standard, US$199; Professional, US$299.
Phone: (800) 965-2965
Web Site: http://www.woll2woll.com/1stclass

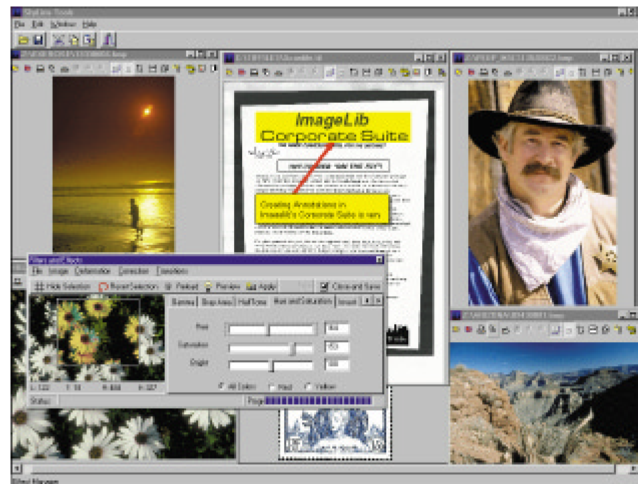


SkyLine Tools Announces ImageLib Corporate Suite 5.0 for Delphi 5

SkyLine Tools Imaging announced the release of *ImageLib Corporate Suite 5.0 for Delphi 5*, the company's imaging solution for Delphi 5 developers.

Features of the newest version of the ImageLib Corporate Suite include upgraded memory for larger images produced by newer digital cameras; upgraded TWAIN scanning that meets the specifications for newer scanners; a features package that allows annotations to be customized by the developer; and a magnifying glass feature, which allows the user to zoom in to a specific area of the image rather than zooming in to the entire page, as well as lets the user move the magnifying glass around on the screen, continuing to magnify only the area under the "glass."

In addition, this toolkit contains all the features from previous versions, such as anti-aliasing, multi-page TIFF files, and over 40 image-manipulation effects and filters.



SkyLine Tools Imaging
Price: US$599
Phone: (800) 404-3832
Web Site: http://www.imagelib.com

DBI Technologies Announces Solutions::PIM Professional 2.0

DBI Technologies Inc. announced the release of *Solutions::PIM Professional 2.0*, designed to allow developers to build Microsoft Outlook-style appointment-based information management applications. Version 2.0 allows the presentation of appointments by day, week, month, year, or developer definition with the new Virtual Calendar. It also features a scrolling calendar with definable start and end dates, as well as drag-and-drop appointments. You can identify appointments using colors or pictures, or print calendars in multiple views using various paper sizes with a single line of code.

This version offers developers definable security features at multiple levels. Manage the addition of appointments with advanced conflict checking, including a new appointment interval-offset property. Or make the calendar interactive via the Web. All components have been made Internet-ready, making them compliant with Web page scripting.

DBI Technologies Inc.
Price: US$489 (includes 17 32-bit licensed ActiveX components; fully functional sample applications with source code; online tutorials; and comprehensive online help).
Phone: (800) 670-8045 or (204) 985-5770
Web Site: http://www.dbi-tech.com

Delphi
TOOLS

New Products and Solutions

Tabdee Announces TurboSync 1.0

Tabdee Ltd. announced the release of *TurboSync 1.0*, a set of VCL components and Delphi classes that enable Palm conduits to be developed in Delphi.

Conduits are DLLs that handle the synchronizing of data between Palm devices and PCs. TurboSync components allow developers to define the Palm database structure visually in the IDE, and use familiar Delphi idioms to access the data — so there's no more struggling with VC++ and inheriting from poorly documented classes.

Tabdee Ltd.
Price: US$60 (includes full source code for the components, translations of the major conduit APIs, and free minor version upgrades).
Phone: +44 (0) 118 9882561
Web Site: http://www.tabdee.ltd.uk

InstallShield Announces InstallShield for Windows Installer 1.5

InstallShield Software Corp. announced *InstallShield for Windows Installer 1.5*, the latest version of the company's comprehensive setup solution for developers of Windows 2000 logo-compliant applications.

This new version offers Internet-enabled update patching, which allows users of Microsoft's Windows Installer service to create "software patches" of their Windows Installer-based applications for distribution over the Internet. The .MSI patching capabilities allow setup authors to build software patches — updates to an existing installation or set of installations — that contain only differences between the installations, not the complete file set. The update is created as the difference (file- or byte-level) between two releases of the product.

InstallShield for Windows Installer allows developers to write and debug custom actions using the InstallScript language, without leaving the installation development environment. Developers can also reuse existing or previous InstallScript to take advantage of prior setup investment.

InstallShield for Windows Installer 1.5 delivers a GUI-driven setup-authoring environment and an assortment of wizards and help features to shorten development time and decrease the learning curve associated with using the Microsoft Windows Installer service.

Additional features include the ability to automatically include updated COM information in a project at build time; the ability to programmatically add/-remove features, components, merge modules, registry entries, and shortcuts; dynamic file linking with support for subfolders; the ability to call any exportable DLL function and define the required parameters with a simple wizard; an enhanced dialog editor; and additional language support.

InstallShield Software Corp.
Price: US$995 (sold as part of the InstallShield Professional 2000 bundle).
Phone: (800) 374-4353
Web Site: http://www.installshield.com

ProWorks Releases Flipper Graph Control 2.0

ProWorks LLC announced the release of ***Flipper Graph Control version 2.0***, an upgrade to the company's ActiveX charting control. Version 2.0 features include enhanced compatibility with the Web, increased flexibility for scientific and financial charts, an improved look and feel, and greater customizing capability.

Flipper Graph Control 2.0 can download saved data across the Internet, as well as easily integrate into ASP or HTML Web pages, enabling users to manage and display off-site data. Included with version 2.0 is a signed .CAB file, permitting client machines to download a run-time version of the control for use on a Web page.

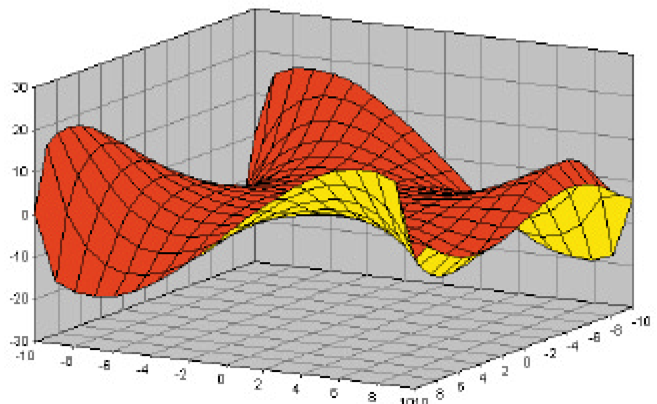
Flexibility has been added to scientific and financial graphs, with the addition of new functions for polynomial curve fitting, moving average, and setting axis data aspect ratio.

The look and feel of version 2.0 has been enhanced with gradient fills for 2D bar charts, a variety of fill style patterns for all chart types, and the ability to place symbols from fonts as marks on a graph. Context menus with editing dialog boxes provide an effortless user interface at run time.

Other new features include ADO Recordset reading, OLE drag-and-drop support, tool tips, and improved date formatting and incrementing.

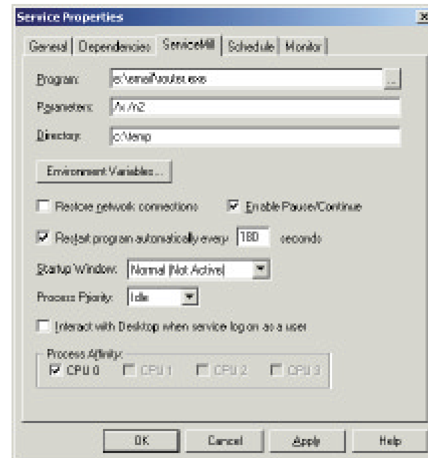
Graph types include 2D and 3D line, bar, points, areas, stacked bars, hi-lo, pie, bubble, spider, and true 3D surfaces. Multiple graphs can be added to the control for drill-down functions or merged to present data on more than two *y* axes. The user can find objects and graph items with the mouse, causing events to be fired to the developer. Creating graphs continues to be fast with auto-scaled axes, numerous examples, and a full tutorial.

ProWorks LLC
Price: US$349
Phone: (541) 752-9885
Web Site: http://www.proworks.com



Active+ Software Announces ServiceMill 3.1.35

Active+ Software announced the release of *ServiceMill 3.1.35*, the system utility that instantly turns any DOS, Windows or OS/2 executable, as well as batch files, into a Windows NT/2000 service.

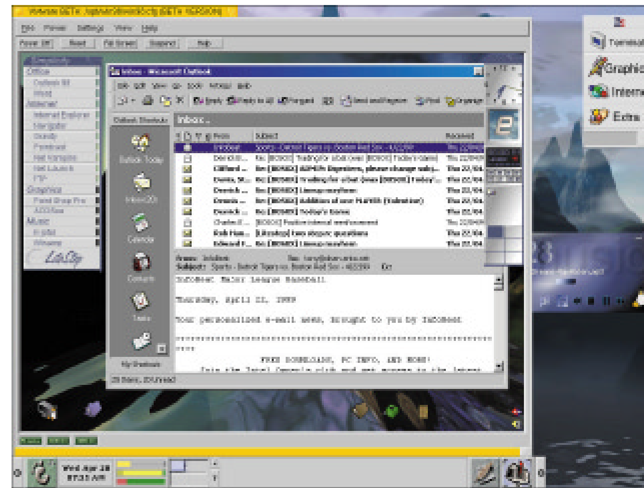


ServiceMill now supports Windows 2000 Kernel Job Objects. When a program is started as a service, the program process will be attached to a Job object. Therefore, if the program starts other processes, they will all run in the same Job. Stopping the service will cleanly stop all processes in the Job, not only the primary process. Another advantage of running a service program through a Job is that all processes running in it will be restricted to a predefined scheduling priority.

Active+ Software
Price: US$65 (single-machine license for Windows NT Workstation/2000 Professional); US$125 (single-machine license for Windows NT Server/2000 Server).
Phone: +33 468054774
Web Site: http://www.activeplus.com

VMware Announces VMware Express for Linux

VMware, Inc. announced the release of its latest desktop product, *VMware Express for Linux*, which was designed for Linux users who want to run Windows 95/98 to access the Windows applications they need.

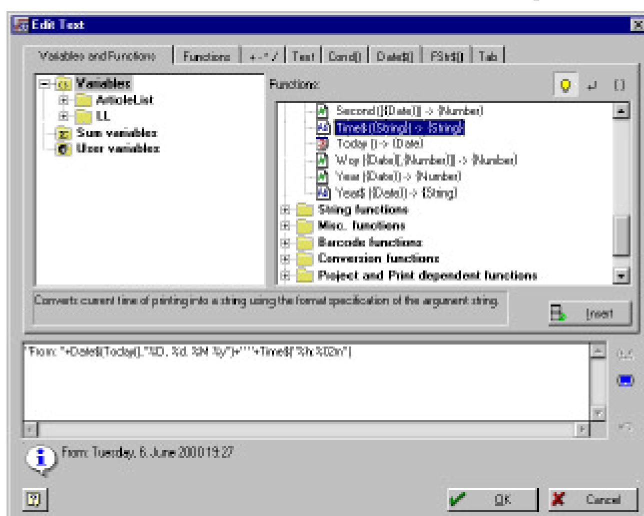


VMware Express for Linux allows users to run Windows 95 or 98 in a VMware virtual machine on top of a Linux host system. A VMware virtual machine works just like a full PC, with full networking and multimedia support. In addition, users will easily be able to share files between the Linux host system and the Windows virtual machine, using the Samba file-sharing service that comes preconfigured.

VMware, Inc.
Price: Call for pricing.
Phone: (650) 475-5000
Web Site: http://www.vmware.com

combit Announces List & Label 7.0

combit GmbH announced the newest version of its report generator, *List & Label 7.0*, which offers an increased level of user friendliness and a variety of new features. As with the previous version, the List & Label Designer and Internet/e-mail modules can be passed on to the end-user royalty free.



Developers can integrate the report generator in existing applications with a few lines of code. Applications are equipped with functions for creating reports, lists, forms, and labels. Code examples in various programing languages are available to simplify development. The tool is available for all DLL-capable programming languages. Special versions, which only work with Delphi or Visual Basic, are also available.

List & Label consists of a print engine and a form designer. Data is transferred directly from the application, so the tool is independent of a specific database.

The Unicode/Multibyte module can process most character sets, including Asian.

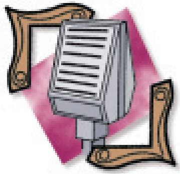
The DTP form designer offers encompassing layout tools and many filter and layout options. These can be used via drag-and-drop. The formula assistant enables complex calculations and string manipulations directly at run time. The integration of RTF text, graphics, and barcodes is also possible, as well as individual printer control for the first and following pages.

combit GmbH
Price: Contact combit for pricing information.
Phone: +49 7531 906010
Web Site: http://www.combit.net/

News

L I N E

October 2000



Inprise/Borland Introduces Borland CLX

New York, NY — Inprise/-Borland announced Borland CLX (component library for cross-platform), the next-generation component library and framework for developing native Linux and Windows applications and reusable components. CLX simplifies graphical user interface, database, and Web application development with a cross-platform component framework design based on the Delphi and C++Builder visual component library (VCL).

CLX will be available in the forthcoming Linux version of the Borland Delphi and C++Builder tools, code-named "Kylix." The Kylix project is a native rapid application development environment for the Linux operating system, and will also be incorporated into the next Windows versions of Delphi and C++Builder.

Borland CLX offers true native Windows and Linux component libraries for ultra-high performance and native services; an object-oriented component framework for building reusable components in Delphi or C++; simplified migrating of Delphi and C++Builder applications to Linux; integration with the next generation of Borland visual development tools for Windows and Linux platforms; and support for commercial, proprietary, and open-source licensed development. Borland CLX also allows developers to leverage existing Delphi and Visual Basic skills.

For more information, visit http://www.inprise.com.

Inprise/Borland Announces JBuilder Support for Apple's Mac OS X

San Diego, CA — Inprise/-Borland announced plans to provide its JBuilder pure-Java development environment for Apple's next-generation operating system. JBuilder's support for Apple's new Aqua graphical user interface lets Java developers build applications on the Macintosh platform that take advantage of the Aqua design elements.

Mac OS X will support the Java 2 Platform, Standard Edition (J2SE), including the Java Hot-Spot Client Virtual Machine, for optimal performance. Using the JBuilder Java IDE, developers will build Java applications running on Mac OS X and realize the benefits of the Aqua look-and-feel, designed to advance the ease-of-use of personal computer user interfaces.

JBuilder for Mac OS X will be available in conjunction with Apple's planned release of Mac OS X early next year. To learn more, visit Inprise/Borland at http://www.borland.com, the community site at http://community.borland.com, or call the company at (800) 632-2864.

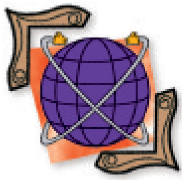
Inprise/Borland's Kylix to Support Apache Application Development

Scotts Valley, CA — Inprise/-Borland announced support of native database-driven Apache Server applications in its forthcoming Linux developer tool set, Kylix. The Kylix project, planned to be available later in 2000, will enable Web application development for the Apache Web Server on the Linux operating system, and for the Windows platform in an upcoming version of Delphi.

By supporting Apache Server applications, the Kylix project will allow developers to build native applications in an object-oriented, component-based development environment, with Linux OS and Windows OS cross-platform flexibility. In addition to supporting the Apache Server, the Kylix project will create a migration path from other HTTP Web servers to the Apache Server for applications developed with Borland Delphi and C++Builder. Web servers supported include Microsoft Internet Information Server (IIS), Netscape servers through Common Gateway Interface (CGI), Internet Server Application Program Interface (ISAPI), and Netscape Server Application Programming Interface (NSAPI).

The Kylix project is intended to be the first high-performance rapid application development tool for the Linux platform.

table of CONTENTS



On the 'Net

CDO / Microsoft Outlook / Microsoft Exchange / Delphi 4, 5

By Kristen Riley

Internet Messaging Made Easy

Getting Started with Collaborative Data Objects

Collaborative Data Objects (CDO) provides a way to include messaging functionality in applications. It does this by providing a simplified, but limited, interface to the underlying MAPI (Messaging Application Programming Interface) library. Since CDO is a COM object that takes the form of a dynamic-link library (DLL), the properties and methods of CDO can easily be accessed with Delphi after importing the DLL as a type library.

There are two parts to CDO: the main CDO library, and the CDO rendering library. The main CDO library provides the ability to send and receive mail messages, interact with folders and address books, and generate meeting items and appointments. The CDO rendering library is used to render CDO objects and collections into HTML for use on the Web. This article introduces Delphi developers to the main CDO library with example applications that retrieve address lists, retrieve Inbox e-mail, and send e-mail.

Getting Started with CDO

Microsoft Outlook (97, 98, or 2000) must be installed on your machine to use CDO. If Outlook 98 or Outlook 2000 is installed, make sure they were installed with the Corporate or Workgroup Mode to ensure that the proper underlying MAPI files were also installed. A message store is also necessary, preferably Microsoft Exchange Server.

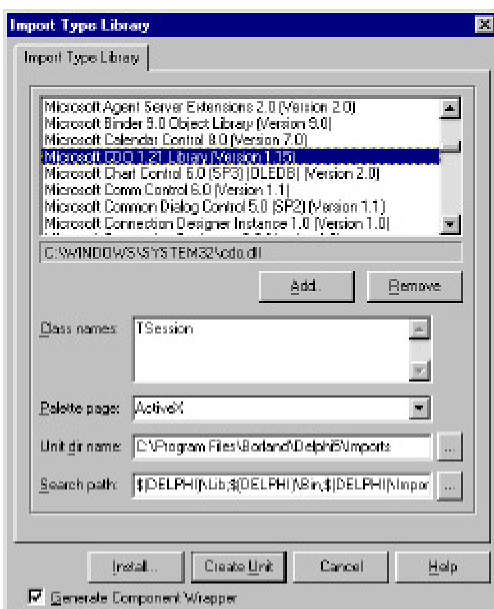


Figure 1: The Import Type Library dialog box.

The latest version of CDO is 1.21. However, there are different versions for Windows 95, 98, and NT. If cdo.dll isn't already on your machine, copy the correct version of it into your \Windows\System32 directory. The CDO files can be downloaded from http://www.microsoft.com/exchange/downloads/CDO.htm. Register the file with the regsvr32.exe cdo.dll command from your \Windows\System32 directory.

To use CDO in Delphi applications, first import the library. To do this, click **Project | Import Type Library** to open the Import Type Library dialog box (see Figure 1). (Note: The library may already be installed, so it's a good idea to check the Open Type Library dialog box for its presence before trying to install it.) Click on the **Add** button to open the Open Type Library dialog box. Find where you put your copy of cdo.dll, highlight it, and click the **Open** button. **MAPI (Version 1.21)** should now be highlighted at the top of the Import Type Library dialog box, with **Class names** containing **TSession**. Click the **Create Unit** button. This will create the file MAPI_TLB.pas in the Delphi \Imports directory, e.g. \Program Files\Borland\Delphi5\Imports. When requested, recompile the package and save it.

Now let's look at the file we've created. Open the MAPI_TLB.pas file in Delphi. Search for the first occurrence of the string `_Session = dispinterface;`. As stated in the MAPI_TLB file, you should be looking at the "forward declarations of interfaces defined in the type library." This is a list of the structures used in CDO, and defined further down in the file. The structures we'll use in the examples are: *_Session*, *AddressLists*, *AddressList*, *AddressEntries*, *Messages*, *Message*, *Recipients*, *Recipient*, *Attachments*, and *MessageFilter*.

Now, search for the second instance of the string `_Session = dispinterface;`. This will go to where the *Session* object is defined. All the structures of CDO are accessed beginning with the

Session object. Following the session's structure are the rest of the CDO structures and their properties and methods. As you can see, the properties and methods of these structures are all accessed with variant parameters.

As mentioned, three CDO examples are illustrated in this article: 1) Listing our recipient groups (address books) and their corresponding recipients; 2) Gaining access to the Inbox and sorting, filtering, and listing received e-mails; and 3) Sending e-mail.

First, let's describe the variables and code necessary to add a CDO session to your application — and then log on using that session — since this is common to all three examples.

To use CDO in your application, add MAPI_TLB to your **uses** clause. Then, for the purposes of this article, add a session variable to your **private** declarations:

```
private
  fSession: _Session;
```

For these examples, also add the following variable to the **private** declaration:

```
fConnected: Boolean;
```

Now you can create and refer to the session in your code. In this case, I create the *Session* object when the form is created:

```
if fSession = nil then
  fSession := Session(CoSession.Create);
```

Mailstore Logon

Once your session is created, you can log on, which will connect your application to the mailstore. The *Logon* method is defined as follows:

```
function Logon(ProfileName: OleVariant;
  ProfilePassword: OleVariant; ShowDialog: OleVariant;
  NewSession: OleVariant; ParentWindow: OleVariant;
  NoMail: OleVariant; ProfileInfo: OleVariant):
  OleVariant; dispid 119;
```

There are two ways to log on to the mailstore. The first uses your profile and password; the second uses your server, login, and password. I was forced to use the second way because the first method didn't work with a modem connection (at least in our system). For these example applications, all three fields can be entered, but the second logon method is used only if there is text in the server field. Figure 2 shows the code that fills in the variables for the *Logon* function call (*ProfileName*, *ProfilePassword*, and *ProfileInfo* are declared as strings).

```
if Length(ServerEdit.Text) <= 0 then
  begin
    ProfileName := MailBoxEdit.Text;
    ProfileInfo := EmptyParam;
  end
else
  begin
    ProfileName := '';
    ProfileInfo :=
      ServerEdit.Text + #10 + MailBoxEdit.Text;
  end;

ProfilePassword := PasswordEdit.Text;
```

Figure 2: Assigning values to the variables for the *Logon* function.

The *ProfileName* field is an Outlook profile. This is the name you use when normally logging on to Outlook. However, if you log in using the second method, this field is set to an empty string, and the *ProfileInfo* field is used. The *ProfileInfo* field will then consist of a server and log in name separated by the ASCII linefeed character.

Note that the *ProfileInfo* field is set to *EmptyParam* in the first login. *EmptyParam* works as a parameter placeholder in COM. It tells the COM object to use the default parameter value since the field isn't used. Because the *ProfileInfo* field is the last parameter of the *Logon* function, it could be left off the argument list. However, if *ProfileInfo* was the second parameter, *EmptyParam* would have to be used so that the next parameter, *ProfilePassword*, would be called in the correct place.

The next parameter is the password. If you're already logged in to Outlook, and you're going to log in under the same profile in your application, the password doesn't seem to be necessary.

The *ShowDialog* parameter, when set to True, will open the Outlook logon dialog box if the profile/server/password information was searched and deemed incorrect. If this is set to False and incorrect logon information is supplied, a temporary profile will be created. However, the temporary profile won't have an Inbox — a fact you can put to use when automating applications. This will be examined later when looking at the Inbox example application.

The *ParentWindow* parameter is set to -1, so that any Logon popup dialog boxes aren't linked to a parent window. The *NoMail* and *NewSession* variables are set to True.

Now, wrap the *Logon* call with an exception handler. Even though the *Logon* is supposed to return a value, in my testing, it always returned NULL whether or not the login information was valid:

```
try
  fSession.Logon(ProfileName, ProfilePassword, True, True,
                 -1, True, ProfileInfo);
  fConnected := True;
except
  on E: Exception do
    ShowMessage('Logon Failed: ' + E.Message);
end;
```

After calling the *Logon* function, provided there are no exceptions (*fConnected* will be True) and assuming the logon information given was correct (see the Inbox example for more detail), then you can start working with the other properties of the *Session* object. For instance, *fSession.CurrentUser* contains the logged-in user's name, and *fSession.Name* contains profile account information. The following examples will be using the session's *AddressLists*, *Inbox*, and *Outbox* properties.

Recipient Listing Example

Figure 3 shows the hierarchy of the structures we'll use to list stored recipients.

```
Session
  AddressLists
    AddressList
      AddressEntries
        Recipient
```

Figure 3: Object hierarchy used in first example.

First, access the session's list of address lists. This is done using the *AddressLists* property of the session. The *AddressLists* structure

```
RecipientListBox.Clear;

I := AddressListBox.ItemIndex;
Inc(I);

MyAddressItem := fSession.AddressLists.Item[I];
AddEntries := MyAddressItem.AddressEntries;

for I := 1 to AddEntries.Count do begin
  MyRecipientItem := AddEntries.Item[I];
  RecipientListBox.Items.Add(MyRecipientItem.Name + ', ' +
                             MyRecipientItem.Address);
end;
```

Figure 4: Adding recipient information to the *RecipientListBox*.

```
Session
  Inbox
    Messages
      MessageFilter
        Message
          Attachments
            Recipients
              Recipient
```

Figure 5: Object hierarchy used in second example.

can be examined by searching for `AddressLists = dispinterface` in the MAPI_TLB file. Each item of the *AddressLists* is an *AddressList* (which is defined after *AddressLists* in the MAPI_TLB file). This example iterates through the items of *AddressLists* and uses a temporary OleVariant variable, *MyAddressItem*, to hold the *AddressList* information. Then, using the *MyAddressItem* variable, it adds the name of each address list to the *AddressListBox*:

```
AddressListBox.Clear;
for I := 1 to fSession.AddressLists.Count do begin
  MyAddressItem := fSession.AddressLists.Item[I];
  AddressListBox.Items.Add(MyAddressItem.Name);
end;
```

Note that the *AddressLists* count goes from 1 to *n*. Because the count for this property is relatively static, the placement of the *AddressList* item is used later to obtain additional information on the item. The *AddressList* item will have an *AddressEntries* property (see *AddressEntries* in the MAPI_TLB file). The items of the *AddressEntries* are the recipients to be listed. The structure of a recipient can be found by searching for `Recipient = dispinterface` in the MAPI_TLB file. Using the OleVariant variables *MyAddressItem*, *AddEntries*, and *MyRecipientItem*, the names and addresses of the recipients contained in the selected address list are added to the *RecipientListBox* (see Figure 4).

The value returned by *AddressListBox.ItemIndex* was incremented because the listbox has a 0 index and *fSession.AddressLists* starts at 1. The *AddEntries* count also begins at 1.

Inbox Access Example

Figure 5 shows the hierarchy of the structures we'll use to list received e-mail messages. In this example, the Inbox folder will be accessed after logging on to the mailstore.

In this case, however, there's a gotcha. Say you're running an automated application in which you want the Logon dialog box popping up on an invalid logon. If you pass the *Logon* function invalid information, an exception won't be raised and a flag doesn't signal this incorrect logon. Instead, a temporary profile will be assigned. As mentioned previously, this temporary profile won't have an Inbox. This means that as soon as you try to access the Inbox, an exception will be raised. You can trap this error (as

```
try
  fSession.Logon(ProfileName, ProfilePassword, True, True,
                 -1, True, ProfileInfo);
  fConnected := True;
  try
    InBox := fSession.InBox;
    fValidInbox := True;
  except
    on E: Exception do
      ShowMessage('Invalid Inbox: ' + E.Message);
  end;
except
  on E: Exception do
    ShowMessage('Logon Failed: ' + E.Message);
end;
```

Figure 6: Raising an exception if the Outlook Inbox isn't present.

shown in Figure 6) where *Inbox* is a local OleVariant variable, and *fValidInbox* is a Boolean variable declared in the **private** section of the form.

By encasing the logon in a **try..except** block, and then accessing the Inbox within another **try..except** block, you can determine whether there was a successful connection to the message store, as well as whether the login information was valid.

After successfully accessing the Inbox, the list of messages in the Inbox can be retrieved (see "Messages = dispinterface" in the MAPI_TLB file). They can also be filtered and/or sorted. The following code is an example of filtering out previously read messages, and then sorting the remaining messages based on the time they were delivered. *Inbox*, *InMessages*, and *MessageFilter* are all OleVariant variables:

```
Inbox := fSession.Inbox;
InMessages := InBox.Messages;
MessageFilter := InMessages.Filter;
MessageFilter.UnRead := 1;  // Only access unread messages.
InMessages.Sort(CdoDescending, CdoPR_DELIVER_TIME);
```

To filter the Inbox messages, first access its *MessageFilter*. Then set the filter to one of the available values, as defined in the *MessageFilter* structure shown in the MAPI_TLB file under "MessageFilter = dispinterface." In this case, any messages that have been read are filtered out, but other filters, such as sender, size, or subject, can be used.

Sorting the messages is done with the message's *Sort* function, defined as:

```
function Sort(SortOrder: OleVariant;
  PropID: OleVariant): OleVariant;
```

The *SortOrder* parameter is simply none, ascending, or descending, corresponding to *CdoSortOrder* (see MAPI_TLB): *CdoNone*, *CdoAscending*, and *CdoDescending*. The *PropID* parameter defines what the messages will be sorted by. In this example, it's delivery time, but many more are listed under *CdoPropTags* in the MAPI_TLB file.

Now that the Inbox messages are sorted, the individual messages can be retrieved. There are two ways to do this. The first is to simply get the count of messages in the Inbox and iterate from 1 to that count (*InMessages.Count*). The second is to use the message's *GetFirst* and *GetNext* functions. When using this second method, every message retrieved must be checked to see if it's valid by passing it to the *VarIsNull* and *VarIsEmpty* functions. If either of these functions comes back positive, then the last message in the list was already retrieved.

```
Recipients := SingleMessage.Recipients;
for I := 1 to Recipients.Count do begin
  SingleRecipient := Recipients.Item[I];
  if SingleRecipient.Type = CdoTo then
    ccstr := 'To: '
  else
    ccstr := 'Cc: ';
  RecipientsListBox.Items.Add(ccstr +
    string(SingleRecipient.Name) + '; ' +
    string(SingleRecipient.Address));
end;
```

Figure 7: Distinguishing mail recipients.

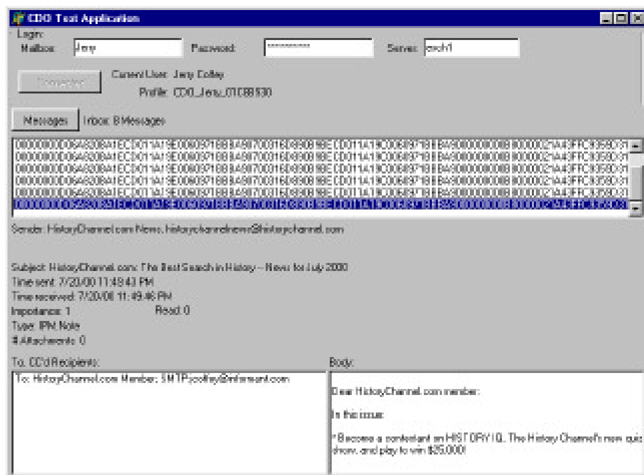


Figure 8: The first demonstration application allows login and lists messages in the Inbox.

The following code shows the Inbox messages being retrieved and then added to a listbox with their IDs as text (*SingleMessage* is an OleVariant):

```
SingleMessage := InMessages.GetFirst;
while not (VarIsNull(SingleMessage) or
      VarIsEmpty(SingleMessage)) do begin
  MessageListBox.Items.Add(string(SingleMessage.ID));
  SingleMessage := InMessages.GetNext;
end;
```

If desired, the information associated with each message could be retrieved in the loop. However, because the ID of each message is actually a GUID and unique to the mailstore, for this example, the ID will be used to retrieve the message contents later, from the session itself, when the listbox is clicked:

```
GUIDstr := MessageListBox.Items[MessageListBox.ItemIndex];
SingleMessage := fSession.GetMessage(GUIDStr, EmptyParam);
```

Once a message is assigned to an OleVariant variable, *SingleMessage* in this case, its properties can be accessed. The *Message* structure can be viewed in the MAPI_TLB under "Message = dispinterface." So, to show the message's subject, you would use:

```
SubjectLabel.Caption :=
  'Subject: ' + string(SingleMessage.Subject);
```

Subject, time sent, time received, body, size, as well as many other values associated with the message, can be easily retrieved in this manner. Other values, such as importance, are simply translated using the constants listed in the MAPI_TLB file. Still others, such as recipients and attachments, require a little more work, but

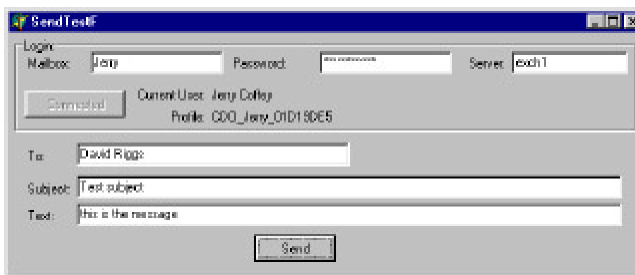


Figure 9: The second demonstration application uses CDO to send messages.

aren't difficult to reach. As in the first example, in which recipients were contained and accessed from an *AddressList*, the recipients of a message are contained in its *Recipients* property. But in this case, the recipients can be distinguished as directly mailed or CCd (see Figure 7).

```
Session
  Outbox
    Messages
      Message
        Recipients
```

Figure 10: Object hierarchy used in third example.

I must mention something now, before delving into the last example on sending mail: the message type property. When running this example, the message type will be "IPM.Note" for most messages received. For Outlook to easily handle e-mails, make sure that any e-mails sent by your application have their corresponding type declared as "IPM.Note."

I should also mention that two example applications accompany this article, and are available for download (see end of article for details). The first demonstrates the techniques we've discussed so far, and is show in Figure 8.

Send Mail Example

This last example will briefly demonstrate how to send e-mail via CDO. The example application is shown in Figure 9. Figure 10 shows the structures involved.

In this example, the logon process is similar to that of the Inbox example, but checks for a valid Outbox. After this, once the needed information to send is ready, create a new message with the message's *Add* function:

```
function Add(Subject: OleVariant; Text: OleVariant;
  Type_: OleVariant; Importance: OleVariant): OleVariant;
```

```
ANewMessage := fSession.OutBox.Messages.Add(
  SubjectEdit.Text, TextEdit.Text, 'IPM.Note', CdoHigh);

if (VarIsNull(ANewMessage) or
    VarIsEmpty(ANewMessage)) then
  Exit;
try
  ANewMessage.Recipients.Delete;
  ANewMessage.Recipients.AddMultiple(ToEdit.Text, CDOTo);
  // Bring up dialog if bad address.
  ANewMessage.Recipients.Resolve(True);
  ANewMessage.Update;
  ANewMessage.Send(True, False, 0);
except
  on E: Exception do
    ShowMessage('Send Failed: ' + E.Message);
end;
```

Figure 11: Remaining code to send an e-mail.

The *Subject* and *Text* parameters are the subject and text of the e-mail message to be sent. *Type_* is the string "IPM.Note" as previously described. *Importance* is one of the *CdoImportance* constants (*CdoLow*, *CdoNormal*, *CdoHigh*) defined in the MAPI_TLB file. To begin, call the *Add* function and place the result into an OleVariant variable, *ANewMessage* in this case. Figure 11 shows the rest of the code needed to send an e-mail.

Before sending the e-mail, clear any recipients that were previously listed. Then add the sending e-mail recipient to the message's recipients with the *AddMultiple* function. After adding the recipients (with *CDOTo*, *CDOCc*, or *CDOBcc*), use the *Resolve* function to check that the e-mail addresses given are in the proper format. If True is passed to the *Resolve* function, a dialog box will pop up if any of the given addresses are badly formed. Then update the message.

The *Send* function sends the e-mail. The *Send* function is defined as:

```
function Send(SaveCopy: OleVariant; ShowDialog: OleVariant;
  ParentWindow: OleVariant): OleVariant;
```

Setting the *SaveCopy* parameter to True will put a copy of the sent message into the sender's "Sent Item" mailbox. The *ShowDialog* parameter, when True, will cause an extra dialog box to pop up on sending. Note that this dialog box will cause an exception if cancelled. The *ParentWindow* can be set to 0, or a valid window handle, for the *ShowDialog*'s dialog box to use.

Conclusion

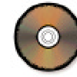
Although these examples have been brief, I hope they have given you a taste of how to use CDO to build mail-enabled applications. Although there's plenty of information about CDO that can't be covered in this article, I recently bought an excellent book on CDO called *Professional CDO Programming* (listed at the end of this article). If you need to develop CDO applications, I highly recommend this book. Although it's aimed at C and Visual Basic programmers, once you understand how to use CDO with Delphi, it isn't difficult to figure out how to translate the COM functionality for use with Object Pascal. ∆

Bibliography

- *ADSI CDO Programming with ASP* by Mikael Freidlitz and Todd Mondor [Wrox Press, 1999].
- *Professional CDO Programming* by Dan Mitchell, Siegfried Weber, and Donald Xie, [Wrox Press, 1999].
- White paper: *Collaborative Data Objects: Using E-Mail in Your Application* by Dr Bruce E. Krell and Ken Miller (http://msdn.microsoft.com/library/techart/collabdataobjs.htm).

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\OCT\DI200010KR.

Kristen has been programming Windows applications for over 10 years. She works at Vast Solutions, Inc., a pioneering wireless application integration and outsourcing company. She can be contacted there at Kristen_Riley@Vast.com.



By Ron Gray

Automating Word

Part II: Word Components in Delphi and Using OLE

Microsoft Word offers many features beyond simple word processing. Using Automation and OLE (Object Linking and Embedding), any Delphi application can easily integrate Word functionality. Last month, the first installment of this two-part series examined Automation in Delphi and looked at the Word object model. This installment presents the Word components available in Delphi 5, and looks at how to link and embed documents using OLE.

Delphi 5 Word Components

Before describing the Word components, consider how compile-time Automation is handled. Word provides type information about its objects, methods, and properties in a type library (msword8.olb for Word 97, msword9.olb for 2000). Delphi provides the ability to import the file and create wrapper classes to the interfaces.

The type library defines the Word *Application* object, which is the top-level object through which all other objects are available. However, the library also defines CoClasses for several other global objects that can be accessed directly by calling the *Create* method of the CoClass client proxy class. In other words, they are stand-alone objects that don't need to go through the *Application* object. For example, rather than using an *Application* object to create a document, the *Document* object can be used directly.

Delphi 5's Word components elegantly wrap the CoClasses, and greatly simplify and encapsulate the Automation process. The following Word components are available in Delphi 5, on the Servers page of the Component palette:

- *WordApplication* represents the entire Microsoft Word application, and is the top-level object of the Word object model.
- *WordDocument* represents a document.
- *WordFont* contains font attributes (such as name, size, color, and so on) for a specified object.
- *WordParagraphFormat* represents all the formatting (such as alignment, spacing, and style) for a paragraph.
- *WordLetterContent* represents elements of a letter.

These components can be used independently, or while connected to each other. Some can be created and applied to properties of other components. For example, to change the paragraph formatting in multiple documents, you could create and format a *WordParagraphFormat* object, then assign it to the *Selection* property of each document.

Sample code presented in Part I of this series called *CoApplication_.Create* to return an *_Application* data type. Using the component, the initialization code changes to this:

```
var
  oWord : TWordApplication;
begin
  oWord := TWordApplication.Create(Self);
  oWord.Connect;
  ...
end;
```

Of course, *TWordApplication* still calls *CoApplication_.Create*. So what has changed? A lot. Not only have the events been wrapped in the components, much of the management work has been added as well. *TWordApplication* and the other Word components descend from *TOleServer* — an abstract class added to Delphi 5 that represents an imported COM server. *TOleServer* provides a framework for connecting to Automation servers, dispatching events, and performing other COM-related work.

Another benefit of the Word components is that many of the common methods have been overloaded to support variable parameters. This allows you to omit unused parameters, so code that used to look like this:

```
oWord.ActiveDocument.PrintOut(EmptyParam, EmptyParam,
  EmptyParam, EmptyParam, EmptyParam, EmptyParam,
  EmptyParam, EmptyParam, EmptyParam, EmptyParam,
  EmptyParam, EmptyParam, EmptyParam, EmptyParam);
```

can omit unused parameters to be written like this:

```
oWord.ActiveDocument.PrintOut;
```

Getting Started

There are many subtleties when integrating Word into a Delphi application, but the basics involve connecting to the Automation server (Word), then manipulating documents.

Establishing a Connection

Before using any of the Word components, a connection must be established between the Delphi application and the Word Automation server. For a number of reasons, the component may fail to connect to the server. Therefore, you should always wrap the connection in a **try..except** statement:

```
try
  WordApplication1.Connect;
except on E: Exception do
  begin
    E.Message := 'Word unavailable.';
    raise;
  end;
end;
WordApplication1.Visible := True;
```

How the connection is established depends on three important properties. First, *AutoConnect* specifies whether the server is loaded when the application first starts. Second, *ConnectKind* determines how the connection to the server is established. It can be set to the following possible values:

- *ckRunningOrNew* attaches to a running server or creates a new instance of the server.
- *ckNewInstance* always creates a new instance of the server.
- *ckRunningInstance* only attaches to a running instance of the server.
- *ckRemote* binds to a remote instance of the server located on the machine for *RemoteMachineName*.
- *ckAttachToInterface* doesn't bind to the server. Instead, the application supplies an interface using the *ConnectTo* method, which is introduced in descendant classes.

Third, *RemoteMachineName* specifies the machine running the COM server, when *ConnectKind* is *ckRemote*.

The application might require a specific connection type. However, to reduce load time, you should connect to a running instance of Word when you can. In applications where documents are heavily manipulated, the *AutoConnect* property can be set to True to load Word when the application starts.

Once the Word component is successfully connected, you can call properties and methods of that object.

Using the *Application* Object

The *Application* object is used mainly to set application-specific attributes (such as the size and appearance of the application window), or to get access to other objects (such as the *Documents* object). The following example uses the *WordApplication* component to set the size of the window:

```
with WordApplication1 do begin
  WindowState := wdWindowStateNormal;
  Caption := 'Integrating Word and Delphi';
  Height := 300;
  Width := 300;
  Left := 0;
  Top := 0;
  Visible := True;
end;
```

When managing multiple documents, it's easier to use the *Application* object, because the *Documents* property already contains a list of all open documents. Assign individual documents to a *WordDocument* object as needed. An earlier example showed how to create a new document with the *Application* object:

```
WordApplication1.Documents.Add(EmptyParam, EmptyParam);
```

The return value of the *Add* method is an interface to the new *Document* object. To manipulate the document, it must be assigned to a variable, such as a *WordDocument* object. The following code uses the *Application* object to create a new document, and assign it to a *WordDocument* object:

```
WordDocument1.ConnectTo(
  WordApplication1.Documents.Add(EmptyParam, EmptyParam));
```

The *ConnectTo* method is used to connect the *WordDocument* object to an interface of the *Application* object, i.e. the Word document.

Working with Documents

You can reuse the same *WordDocument* variable to connect to any document. The following example connects the *WordDocument* to the active document:

```
WordDocument1.ConnectTo(WordApplication1.ActiveDocument);
```

This example connects to a specific document:

```
WordDocument1.ConnectTo(
  WordApplication1.Documents('MyDoc.Doc'));
```

A *WordDocument* object can also be used, without the *Application* object, to manage a single document:

```
WordDocument1.ConnectKind := ckRunningOrNew;
WordDocument1.Connect;
```

To save a new document the first time, use the *SaveAs* method:

```
var
  FileName: OleVariant;
begin
  FileName := 'MyDoc.Doc';
  WordDocument1.SaveAs(FileName);
end;
```

To save an existing document, call the *Save* method.

Working with Text

Word objects exist for manipulating the various elements of a document: characters, words, sentences, paragraphs, and sections. These objects provide access to *Range* or *Selection* objects, which are used to modify text. The *Range* object represents a contiguous

area in a document defined by a starting and ending character position. A *Selection* object represents the selection in a document window pane.

The following example creates a *Range* object at the beginning of the document, and inserts some text (see Figure 1):

```
var
  Text, nStart, nEnd: OleVariant;
begin
  nStart := 0;
  nEnd := 0;
  Text := 'Hey now! ';
  WordDocument1.Range(nStart, nEnd).InsertBefore(Text);
end;
```

This next example marks the second paragraph in bold:

```
WordDocument1.Paragraphs.Item(2).Range.Bold := 1;
```

This example does the same thing, but uses the *WordFont* object:

```
WordFont1.ConnectTo(
  WordDocument1.Paragraphs.Item(2).Range.Font);
WordFont1.Bold := 1;
```

Working with Other Objects

Of course, there are many other objects available in Word for working with text. The *Find* and *Replacement* objects, *Table* and other associated objects, and the *HeaderFooter* object are just a few.

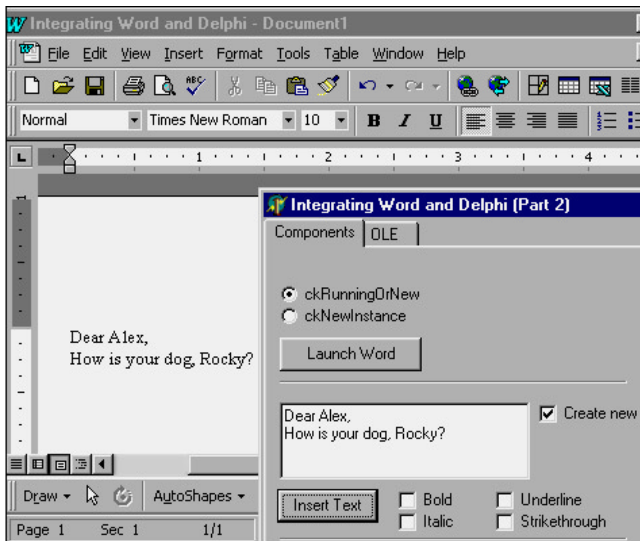


Figure 1: The sample application in action. Here, it changes properties of the *Application* object, then inserts text in a document.

Events

The Word *Application* object publishes three event interfaces, shown in Figure 2. They are exposed in their appropriate components.

These events let the Delphi application monitor a specific document, or set of documents. For example, by responding to the *Close* event, an application can create a separate backup copy, or save the document to a BLOb (Binary Large Object) field.

Object Linking and Embedding

So far, this article has presented ways to automate Word from a Delphi application. This approach manipulates Word as a separate application to control separate, external Word documents. However, to truly integrate Word, the application must be able to link or embed documents. This would be necessary in the following situations:

- The functionality of the Delphi application requires certain Word documents (such as mail merge templates). A safer approach would be to store the documents in BLOb fields, rather than have them external to the application.
- The Delphi application must display the contents of Word documents directly on a Delphi form, without loading Word each time.
- Users want to edit the Word document from within the Delphi application, without launching Word.

These requirements can be satisfied with OLE 2. OLE is closely related to Automation, and is also built on the COM architecture. It enables applications to create compound documents that contain components of other applications. In other words, a Delphi application can store and manage documents created in Word.

Object linking allows you to paste information as a link into an application, and have the data dynamically updated when it is changed in the source application. Double-clicking on the data launches the source application for editing. Object embedding allows you to create compound documents where data from one application can be placed in another application as an object. A Delphi application can display the data of an OLE object, without knowing the application that created it. Double-clicking on the object can initiate in-place activation. This means that, rather than launching the source application for editing, the client application temporarily acts like the source application by changing its menus. This is possible because the object contains all the necessary information to edit itself.

The client application must use a container to host the OLE object. Delphi's *TOleContainer* encapsulates the complexity of embedding or linking OLE objects. There are several ways to get the object into the container. An object can be pasted (using *Paste* or *PasteSpecialDialog*), inserted (using *InsertObjectDialog*), dragged and dropped, loaded from a file (*CreateObjectFromFile*), created directly (*CreateObject*), or streamed (*CreateFromStream*). Delphi's OLE Container samples demonstrate most of these features, so there's no need to describe them here. There are MDI and SDI versions in the \Delphi5\Demos\ActiveX\Olecntrs directory.

Class	Event	Description
Application	*DocumentChange*	Occurs when a new document is created, an existing document is opened, or when another document is made the active document.
Application	*Quit*	Occurs when the user quits Word.
Document	*Close*	Occurs when a document is closed.
Document	*New*	Occurs when a new document based on a template is created.
Document	*Open*	Occurs when a document is opened.
ActiveX	*GotFocus*	Occurs when focus is moved to an embedded ActiveX control.
ActiveX	*LostFocus*	Occurs when focus is moved from an embedded ActiveX control.

Figure 2: Event interfaces in Microsoft Word.

These demonstrations show that Word documents can be created, stored, and edited directly in OLE containers without using the Word com-

ponents. You can even make calls to the *Word* object through the container. The example in Figure 3 uses the OLE container to create a new document, switch to in-place activation, activate the document, then insert some text.

Note that the call to *CreateObject* passes *Word.Document* as the class name to create a new document. The second parameter (False) indicates the document won't be displayed as an icon, but as it would be displayed by the server application. The call to *DoVerb* is made with *ovPrimary*, the default action, to activate the object. Once the object is activated, you can get access to it through the *OleObject* property. The last statement in the previous code uses *OleObject* to return the Word *Document* object. It then uses the *Application* object to insert the text.

Referencing the *Document* object in this way uses run-time Automation, described earlier in this article. The primary disadvantages of this approach are lack of syntax checking at compile time, and slower execution. Before making multiple Automation calls to a container object, consider assigning it to a *WordDocument* object to get the benefits of compile-time Automation. The following example assigns the container object created previously to a *WordDocument* object, and inserts some more text:

```
WordDocument1.ConnectTo(
  IUnknown(OleContainer1.OleObject) as _Document);
WordDocument1.Application.Selection.TypeText(
  'Hey Now! Again!');
```

The object is cast to the *IUnknown* interface of a *_Document* object, which is what the *ConnectTo* method expects.

Saving Documents to Tables

The OleContainer control is used to link or embed OLE objects, such as Word documents. However, this link is not persistent and must be managed by the developer. Likewise, the Word components in Delphi 5 must also be managed in the sense that a mechanism for loading documents must be implemented. If direct links to specific documents must be maintained by the Delphi application, then a common approach is to save the name and location of the external document in text fields, or save the actual document in a BLOb field. Each has advantages and drawbacks.

Saving only the name and location of the document allows other applications to share documents and make edits, and doesn't cause a load on the database. However, when an application relies on external documents, the rule of entropy usually makes them disappear. Saving documents directly to BLOb fields as OLE objects is easy and reduces the risk of losing them, but bloats the database and reduces performance.

Consult your database documentation to see how OLE objects should be stored in tables. For example, BLOb data in InterBase is further defined by a subtype. OLE objects must be stored in a BLOb field with subtype 0, the default used to store binary data. The following SQL statement adds a BLOb field to the CustomerLetter table:

```
ALTER TABLE CustomerLetter
  ADD OleObject BLOB SUB_TYPE 0 SEGMENT SIZE 80
```

TBlobStream is used to access or modify the value of a BLOb field. Word documents are loaded from BLOb fields into *TBlobStream* objects. From here, it can easily be loaded into an OLE container or *WordDocument* object. The example in Figure 4 retrieves a document from a BLOb field, puts it in a container, then activates it.

Note that the stream must be created and freed each time. You should never reuse a BLOb stream. To save the contents of the OLE container back to the BLOb field, use the *SaveToStream* method, as shown in Figure 5.

```
OleContainer1.CreateObject('Word.Document', False);
OleContainer1.AllowInPlace := True;
OleContainer1.DoVerb(ovPrimary);
OleContainer1.OleObject.Application.Selection.TypeText(
  'Hey Now!');
```

Figure 3: A *TOleContainer* object is used with in-place activation and run-time Automation.

```
var
  oStream : TBlobStream;
begin
  oStream := nil;
  try
    Table1.Open;
    try
      oStream := TBlobStream.Create(Table1.FieldByName(
                   'OleObject') as TBlobField, bmRead);
      OleContainer1.LoadFromStream(oStream);
      Olecontainer1.DoVerb(ovPrimary);
    except
      MessageDlg(
        'Document not read successfully from BLOb field.',
        mtWarning, [mbOK], 0);
    end;
  finally
    oStream.Free;
    Table1.Close;
  end;
end;
```

Figure 4: Retrieving a document from a BLOb field, placing it in a container, and activating it.

```
var
  oStream : TBlobStream;
begin
  oStream := nil;
  if OleContainer1.State <> osEmpty then
    try
      Table1.Open;
      try
        Table1.Edit;
        oStream := TBlobStream.Create(Table1.FieldByName(
          'OleObject') as TBlobField, bmReadWrite);
        OleContainer1.SaveToStream(oStream);
        Table1.Post;
        MessageDlg('Document saved to BLOb field.',
                   mtInformation, [mbOK], 0);
      except
        MessageDlg('Document not saved to BLOb field.',
                   mtWarning, [mbOK], 0);
      end;
    finally
      oStream.Free;
      Table1.Close;
    end
  else
    MessageDlg('There is no document to save.',
               mtWarning, [mbOK], 0);
end;
```

Figure 5: Using the *SaveToStream* method to save the contents of the OLE container back to the BLOb field.

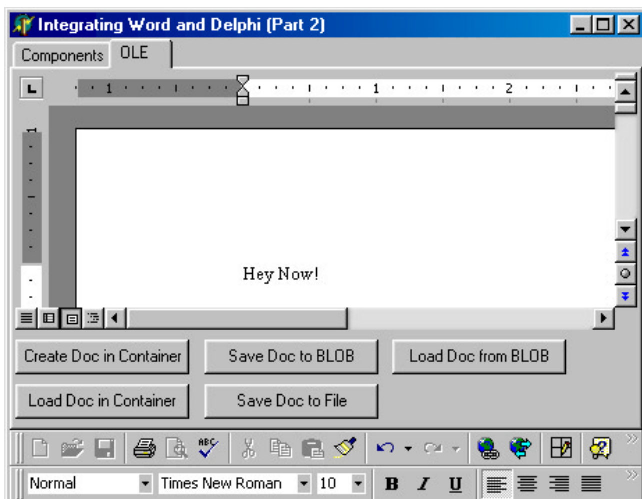


Figure 6: An application demonstrating these techniques is available for download.

This time, the stream was created with read/write capabilities so changes can be saved. The code also verifies that the container isn't empty before saving. A sample application that demonstrates all of the techniques discussed in this article (see Figure 6) is available for download; see end of article for details.

Reverse Role Playing

This article has discussed using a Delphi application as an Automation controller of the Word Automation server. This allows the Delphi application to "control" Word documents. However, because a Delphi application can also be an Automation server, and Word can also be an Automation controller, it's possible to reverse the roles so that Word can control parts of a Delphi application.

Consider a customer-locate dialog box in a Delphi application that can be called from anywhere in the application, and returns the record of the selected customer. By converting the Delphi application to an Automation server, and exposing the customer-locate dialog box through a method, users of Microsoft Word can invoke the locate dialog box to find customers. Once the customer is selected, the name and address can be automatically entered in the document.

Conclusion

Microsoft's Automation and OLE 2 technologies make it possible to integrate Word into Delphi applications. Delphi 5's Word components make it easy. With just a few lines of code, applications can become powerful word processors. However, the uses of Word extend far beyond basic word processing, because there are so many other features available. The benefits can be tremendous. ∆

References

- The Word 97 object model is documented in the Help file vbawrd8.hlp. This file comes with Office 97, but is not installed by default. To install it, run the Microsoft Office 97 Setup program and select **Add/Remove components**. Select **Help for Visual Basic** under the help options. The Word 2000 object model is documented as part of the Microsoft Developer Network, Office 2000 Developer edition, which is installed as part of Office 2000 Developer.
- Microsoft Office development information can be found at http://www.microsoft.com/officedev/.
- See the *Microsoft Office 2000 Visual Basic Programmer's Guide* for detailed information on, and examples of, automating Word 2000.

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\OCT\DI200010RG.

Ron Gray is a software developer specializing in business database applications. He has written numerous articles using different languages and is the author of LookUp Manager, a collection of components for visually managing lookup codes and abbreviations in applications. He can be reached via e-mail at rgray@compuserve.com.

Table of Contents

By Keith Wood

Database Persistent Objects

Part II: Generating *TDBPersistent* Classes with a Wizard

Last month we looked at how to create a class that could automatically store its published properties in a relational database. Now it's easy to create classes that have this ability; we simply derive them from *TDBPersistent* and declare the properties. There are some bookkeeping steps involved, so a constructor and assignment method would be useful. To make it even easier, therefore, we can create a wizard that generates all of this for us.

Delphi Wizards

Delphi wizards (also known as experts) allow us to extend the abilities of the Delphi IDE. They come in four basic types:

1) Project wizards that generate entire programs.
2) Form wizards that build single forms or units (these types appear in the New Items dialog box).
3) Standard wizards that appear on the Help menu.
4) Add-in wizards that have full access to the Delphi environment.

The type dictates how we interact with the wizard and what the expected result is.

All wizards are created as DLLs. They must export a single function, *InitExpert*, which is called by Delphi (when it's loaded) to register the new wizard. Inside this function it calls *RegisterProc*, passing an instance of the expert as a parameter. Thereafter, Delphi can enquire through the expert interface to determine its type and any other necessary details.

Since we're creating a single unit to be incorporated into a larger application, we'll create a form wizard. To begin, we must derive it from *TIExpert* (defined in the ExptIntf unit), which is the base for all experts and wizards. We must then override several of the class' methods to customize it for our use. The following four methods are required for all wizards:

1) *GetStyle* is revised to return the form style (*esForm* for our new wizard).
2) *GetState* indicates that the wizard is available (*esEnabled*).
3) *GetName* returns the new name.
4) *GetIDString* returns a unique identifier. The identifier must be unique world-wide and is usually of the form <Company>.<ExpertName>.

The rest of the methods are optional depending on the style of the wizard. For a form wizard we need the following: *GetAuthor* provides the name of the author, *GetComment* returns a longer description, and *GetPage* indicates on which page in the New Items dialog box the wizard should appear. *GetGlyph* must be overridden to return an image for display in this dialog box. With the Image Editor tool, open the resource file for the project (.res) and add a new icon. Update its appearance and give it a meaningful name before saving the file. Back in the code, we load this resource with the *LoadIcon* function.

Setting the style to a form wizard means it will appear as an entry in the dialog box invoked by the File | New menu item. The previously mentioned methods are then called to determine the appearance of the wizard. Extended details, such as the description and author's name, are shown if we select the View Details option on the popup menu. When we activate the wizard, Delphi calls its *Execute* method to perform its functionality.

Our wizard displays a dialog box to obtain the information necessary to generate the new unit. When the Finish button is clicked, it writes the code out to a file, and then adds that file to the current project.

User Interface

The dialog box presented to the user consists of two pages: one for the class name, parent class,

and the name of the file to be created; and one for the list of properties belonging to the class (see Figures 1 and 2).

The parent class defaults to *TDBPersistent*, but can be replaced with another name. This is done because the new class must eventually derive from this base class to provide the desired abilities of automatic storage in a relational database. Any value may be entered as the name of the new class (an initial "T" is added as necessary), but remember that this also becomes the name of the database table. The output filename defaults to the class name (without the initial "T") plus a .pas extension, but (again) can be overridden. Clicking the **Browse** button allows us to search for an appropriate output location and filename. The Save dialog box also warns us if we plan to overwrite an existing file.

On the second page, we have a string grid containing the list of properties belonging to the new class. Entries are added and removed with the **Add** and **Delete** buttons. Each entry consists of the property name, a flag to indicate whether it's a primary key field, and its type. The spin control following the type name allows the length of strings to be specified and is ignored for all other types. Types can be selected from the drop-down list, or can be entered manually if they're not one of the standard types.

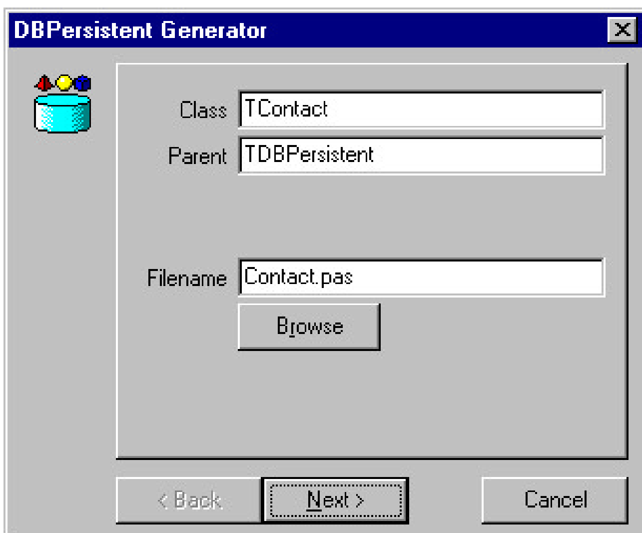


Figure 1: The DBPersistent Wizard in action. First, the class name, parent class, and filename are entered ...

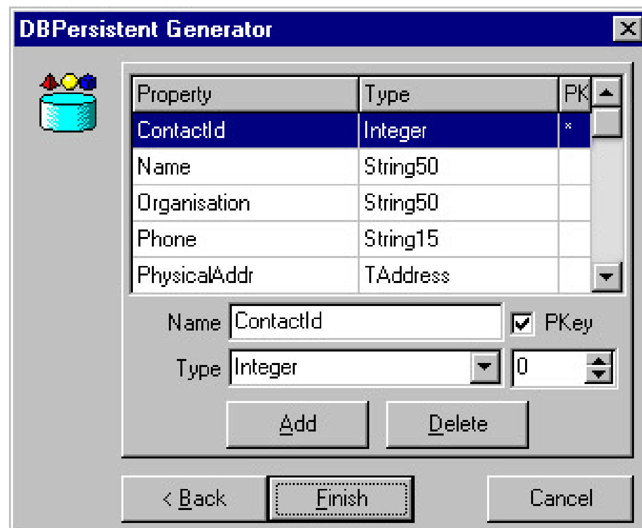


Figure 2: ... then the properties are described.

Generation

Now we have all the details necessary for creating the new unit. Although Delphi provides several classes to assist in generating new form files (using proxies) and their associated units, our needs are much simpler. We're only producing a single-unit file that consists solely of text. Hence we can place the code into a string list, and use its *SaveToFile* method to construct the resulting unit (see Figure 3).

We use a string constant to provide the basic text for the generated unit. This has placeholders for variable sections within it for use with the *Format* function. These are replaced by variable values (for simple substitutions), or by the result of various functions (for more complex code). Examples of the functions can be seen in Figure 4. They provide the list of internal fields for the class, the published properties, and a copy-assignment method, respectively.

The final step in producing the new unit is to incorporate it into the existing project. We can achieve this through the *CreateModule* method of the *ToolServices* object (see Figure 5). Normally this takes as parameters streams that contain the code and graphical (.dfm) parts of a unit, along with a unit name and flags indicating how the other items are to be treated. In our case, we are generating the unit directly to a file, so we don't need to fiddle around with the streams.

By providing the name of the new file, as returned by the *GenerateUnit* method, and an appropriate flag, *cmExisting*, we can leave the stream parameters set to **nil**, and have the unit read from disk. The other flags add the unit to the current project (*cmAddToProject*), and bring the editor window for the newly opened file to the fore (*cmShowSource*).

Installation

The completed wizard is compiled into a DLL. It's suggested that packages are used to reduce the overall size of the final product; instead of a 400KB monster, it can become a svelte 30KB. Move the DLL to an appropriate directory, and then tell Delphi about it. To do this we need to update the registry item containing the list of wizards to load.

```
{ Create the unit file for this persistent object. }
function TfrmDBPersistentGen.GenerateUnit: string;
const
  sPKPrefixes:
    array [Boolean] of string[3] = ('', sPKPrefix);
var
  slsUnit: TStringList;
  sClassName, sParentName, sUnitName: string;
begin
  Result := edtFilename.Text;
  slsUnit := TStringList.Create;
  try
    sClassName := edtClass.Text;
    if sClassName[1] <> 'T' then
      sClassName := 'T' + sClassName;
    sParentName := edtParent.Text;
    if sParentName[1] <> 'T' then
      sParentName := 'T' + sParentName;
    sUnitName := ChangeFileExt(
      ExtractFileName(edtFilename.Text), '');
    slsUnit.Text := Format(sUnitSource,
      [sUnitName, sClassName, GetUses, GetTypes,
       sClassName, sParentName, GetPrivateFields,
       GetPublicAttrs, GetPublishedProps, sClassName,
       GetConstructor, GetAssign, sClassName]);
    slsUnit.SaveToFile(edtFilename.Text);
  finally
    slsUnit.Free;
  end;
end;
```

Figure 3: Generating the unit.

Under the HKEY_CURRENT_USER\Software\Borland\Delphi*n.0*\Experts key in the Windows registry (where *n* is the version number for Delphi), add a new string value. Set its name to DBPGen, and its value to the full path to the DLL we just created. Note that a recompile of the wizard is necessary for each version of Delphi, because the expert interface internals have changed over time.

The next time Delphi is started, we should find a new option in the New Items dialog box. Double-click the DBPersistent Generator icon

```
{ Generate list of internal fields for the properties. }
function GetPrivateFields: string;
var
  iIndex: Integer;
begin
  Result := '';
  with stgProperties do
    for iIndex := 1 to RowCount - 1 do
      if Cells[0, iIndex] <> sDefProperty then
        Result :=
          Result + Format(sFields, [Cells[0, iIndex],
          sPKPrefixes[IsPrimaryKey(iIndex)] +
          Cells[1, iIndex]]);
end;

{ Generate the published properties. }
function GetPublishedProps: string;
var
  iIndex: Integer;
begin
  Result := '';
  with stgProperties do
    for iIndex := 1 to RowCount - 1 do
      if Cells[0, iIndex] <> sDefProperty then
        Result :=
          Result + Format(sProperty, [Cells[0, iIndex],
          sPKPrefixes[IsPrimaryKey(iIndex)] +
          Cells[1, iIndex]]);
end;

{ Generate the copy method. }
function GetAssign: string;
var
  iIndex: Integer;
begin
  Result := Format(sAssign1, [sClassName]);
  with stgProperties do
    for iIndex := 1 to RowCount - 1 do
      Result :=
        Result + Format(sAssign2, [Cells[0, iIndex]]);
  if sParentName = sTDBPersistent then
    Result := Result + sAssign3;
  Result := Result + sAssign4;
end;
```

Figure 4: Selected wizard functions.

```
{ Run the expert and open the resulting file. }
procedure DBPersistentExpert(ToolServices: TIToolServices);
var
  sFilename: string;
begin
  with TfrmDBPersistentGen.Create(Application) do
    try
      if ShowModal = mrOK then begin
        sFilename := GenerateUnit;
        ToolServices.CreateModule(sFilename, nil, nil,
          [cmAddToProject, cmShowSource, cmExisting]);
      end;
    finally
      Free;
    end;
end;
```

Figure 5: Running the wizard.

to invoke it, enter appropriate values, and click the Finish button. A new unit is created and added to the current project. An example unit, corresponding to the entries displayed in Figures 1 and 2, is shown in Listing One.

Conclusion

The database persistence provided by the classes developed last month makes the process of linking objects with a relational database very simple. This month, we've made it even easier by creating a form wizard that extends Delphi, and generates the necessary class definition for us. This, of course, greatly decreases the effort, and the opportunity for introducing coding errors. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\OCT\DI200010KW.

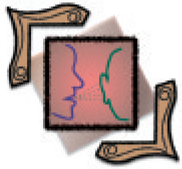
Keith Wood is an analyst/programmer with CCSC, based in Atlanta. He started using Borland's products with Turbo Pascal on a CP/M machine. Often working with Delphi, he has enjoyed exploring it since it first appeared. You can reach him via e-mail at kbwood@compuserve.com.

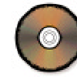
Begin Listing One — *TContact*

```
unit Contact;
{ TContact - database persistent object
  Generated by DBPersistent Expert. }
interface

uses
  Classes, Controls, DBPersist;

type
  String15 = string[15];
  String50 = string[50];

  TContact = class(TDBPersistent)
  private
    FContactId: TPKInteger;
    FName: String50;
    FOrganisation: String50;
    FPhone: String15;
    FPhysicalAddr: TAddress;
    FMailingAddr: TAddress;
    FLastContact: TDate;
  public
    constructor Create(dbmManager: TDBManager;
      ContactId: TPKInteger);
    procedure Assign(Source: TPersistent); override;
  published
    property ContactId: TPKInteger
      read FContactId write FContactId;
    property Name: String50 read FName write FName;
    property Organisation: String50
      read FOrganisation write FOrganisation;
    property Phone: String15 read FPhone write FPhone;
    property PhysicalAddr: TAddress
      read FPhysicalAddr write FPhysicalAddr;
    property MailingAddr: TAddress
      read FMailingAddr write FMailingAddr;
    property LastContact: TDate
      read FLastContact write FLastContact;
  end;

implementation

{ TContact ---------------------------------------------- }
```

```
{ Initialization. }
constructor TContact.Create(dbmManager: TDBManager;
  ContactId: TPKInteger);
begin
  inherited Create(dbmManager);
  Self.ContactId := ContactId;
end;

{ Copy TContact. }
procedure TContact.Assign(Source: TPersistent);
begin
  if Source is TContact then
    with TContact(Source) do begin
      Self.ContactId := ContactId;
      Self.Name := Name;
      Self.Organisation := Organisation;
      Self.Phone := Phone;
      Self.PhysicalAddr := PhysicalAddr;
      Self.MailingAddr := MailingAddr;
      Self.LastContact := LastContact;
      Exit;
    end;
  inherited Assign(Source);
end;

initialization
  RegisterDBPersistentClass(TContact);
end.
```

End Listing One





By Bill Todd

A Quick Way to Shortcuts

A Component for Creating and Modifying Shortcuts

Windows shortcuts provide a way to have as many links to a file as you need — in as many folders as you want. Shortcuts are also the tool for adding a file to the Windows **Start** menu.

In Windows 3.x, creating shortcuts was easy. You had to learn a couple of simple DDE calls, and that was it. In 32-bit Windows, working with shortcuts is more complex, and requires the use of COM and interfaces. This article will look at working with shortcuts in detail, and show you how to build a custom component you can use to create and modify shortcuts in any folder.

```
IShellLinkA = interface(IUnknown) { sl. }
  [SID_IShellLinkA]
  function GetPath(pszFile: PAnsiChar; cchMaxPath: Integer;
    var pfd: TWin32FindData; fFlags: DWORD): HResult;
    stdcall;
  function GetIDList(var ppidl: PItemIDList): HResult;
    stdcall;
  function SetIDList(pidl: PItemIDList): HResult; stdcall;
  function GetDescription(pszName: PAnsiChar;
    cchMaxName: Integer): HResult; stdcall;
  function SetDescription(pszName: PAnsiChar): HResult;
    stdcall;
  function GetWorkingDirectory(pszDir: PAnsiChar;
    cchMaxPath: Integer): HResult; stdcall;
  function SetWorkingDirectory(pszDir: PAnsiChar): HResult;
    stdcall;
  function GetArguments(pszArgs: PAnsiChar;
    cchMaxPath: Integer): HResult; stdcall;
  function SetArguments(pszArgs: PAnsiChar): HResult;
    stdcall;
  function GetHotkey(var pwHotkey: Word): HResult; stdcall;
  function SetHotkey(wHotkey: Word): HResult; stdcall;
  function GetShowCmd(out piShowCmd: Integer): HResult;
    stdcall;
  function SetShowCmd(iShowCmd: Integer): HResult; stdcall;
  function GetIconLocation(pszIconPath: PAnsiChar;
    cchIconPath: Integer; out piIcon: Integer): HResult;
    stdcall;
  function SetIconLocation(pszIconPath: PAnsiChar;
    iIcon: Integer): HResult; stdcall;
  function SetRelativePath(pszPathRel: PAnsiChar;
    dwReserved: DWORD): HResult; stdcall;
  function Resolve(Wnd: HWND; fFlags: DWORD): HResult;
    stdcall;
  function SetPath(pszFile: PAnsiChar): HResult; stdcall;
end;
```

Figure 1: The *IShellLink* interface.

The Interfaces

Shortcuts, or links as they are sometimes called, are actually binary files stored on your hard disk with the .lnk extension. The Windows shell includes a COM object named *ShellLink* for working with shortcuts. The *ShellLink* object implements two interfaces, *IShellLink* and *IPersistFile*, that define the methods for working with shortcuts. Figure 1 shows the declaration of *IShellLink* from SHLOBJ.PAS, and Figure 2 shows the declaration of *IPersistFile* from ACTIVEX.PAS.

Into the *TWinShortcut* Component

The shell of the *TWinShortcut* custom component was created with the Component Wizard in the Object Repository, using *TComponent* as its ancestor. Listing One (beginning on page 23) shows the finished component. To make things easier to find, the properties and their private member variables are in alphabetical order. In the **implementation** section, the methods are divided into three groups: constructor and destructor, property getter and setter, and custom methods. Within each of these groups, the methods are in alphabetical order. The constructor is overridden to automatically create an instance of the *ShellLink* object using the following code:

```
FShellLink :=
  CreateComObject(CLSID_ShellLink) as
  IShellLink;
FPersistFile := FShellLink as IPersistFile;
```

The first statement creates the *ShellLink* object by calling *CreateCOMObject* and passing the *ShellLink* object's class ID as the parameter. The return value is cast to type *IShellLink* to provide a reference to the *IShellLink* interface and its methods. *FShellLink* is a protected member variable of type *IShellLink*. *FPersistFile* is also a protected

```
IPersistFile = interface(IPersist)
  ['{ 0000010B-0000-0000-C000-000000000046 }']
  function IsDirty: HResult; stdcall;
  function Load(pszFileName: POleStr; dwMode: Longint):
    HResult; stdcall;
  function Save(pszFileName: POleStr; fRemember: BOOL):
    HResult; stdcall;
  function SaveCompleted(pszFileName: POleStr): HResult;
    stdcall;
  function GetCurFile(out pszFileName: POleStr): HResult;
    stdcall;
end;
```

Figure 2: The *IPersistFile* interface.

member variable and is of type *IPersistFile*. Casting *FShellLink* to *IPersistFile* provides an interface reference to the *IPersistFile* methods implemented by the *ShellLink* object. *TWinShortcut*'s destructor is overridden, and both *FShellLink* and *FPersistFile* are set to **nil** to destroy the *ShellLink* object. Because COM objects are reference counted, both variables must be set to **nil** before the *ShellLink* object will be destroyed.

You must be able to specify the name and location of the shortcut file you want to work with, and that capability is provided by three properties: *ShortcutFileName*, *ShortcutPath*, and *SpecialFolderLocation*. One big problem in working with shortcuts is figuring out where to create them. For example, if you want to create a shortcut on the user's desktop, you have to know the path to the desktop folder, and that is different for different versions of Windows.

The solution is a Windows API function named *SHGetSpecialFolderLocation*, which takes three parameters. The first is a window handle, which can be set to zero. The second is a constant that identifies the folder you want. To find a partial list of constants, click Start | Programs | Borland Delphi 5 | Help | MS SDK Help Files | Win32 Programmers Reference and search for *SHGetSpecialFolderLocation*. If you have the MSDN Library CD, search for *SHGetSpecialFolderLocation* and you'll find a list of over 40 folder constants. The Win32 Programmers Reference Help file also contains detailed information about *IShellLink* and *IPersistFile* and their methods. The third parameter is a variable of type *PItemIdList*.

After calling *SHGetSpecialFolderLocation*, you will call *SHGetPathFromIdList* to extract the actual path from the *PItemIdList* parameter. The *SpecialFolderLocation* property of *TWinShortcut* is of type Word and corresponds to the second parameter, the folder number, of *SHGetSpecialFolderLocation*. This lets you specify the location of the shortcut by setting the value of the *SpecialFolderLocation* property, or by providing a path in the *ShortcutPath* property.

TWinShortcut has a public *OpenShortcut* method that's used to open an existing shortcut. This method is only three statements long. The first statement is a call to the protected method *GetFullShortcutPath*. *GetFullShortcutPath* returns the full path and filename of the shortcut. The second statement, shown here, actually opens the file by calling the *IPersistFile* interface's *Load* method:

```
OleCheck(FPersistFile.Load(PWideChar(WideString(FullPath)),
  STGM_READWRITE));
```

Load's two parameters are the name of the file and the mode. Because this function is Unicode-compatible, the path must be a null-terminated string of wide chars. Because the call to *GetFullShortcutPath* returns a Pascal ANSI string, the path variable *FullPath* is first cast to type WideString, and then cast to type PWideChar to match the type of the parameter. Note that *Load* is called as a parameter to the *OleCheck* procedure. *OleCheck* examines the value returned by *SHGetSpecialFolderLocation*, and if that value indicates an error, *OleCheck* raises an *EOleSysError* exception. This technique is used for all of the interface method calls in this example, so normal Delphi exception handling can be used to trap errors that occur when using this component. The last line of the *LoadShortcut* method calls the custom method *GetPropertiesFromShortcut*, which calls each of the *get* methods of the *IShellLink* interface and assigns the returned value to the corresponding property of *TWinShortcut*.

Before continuing, let's look at the *GetFullShortcutPath* and *GetPropertiesFromShortcut* methods used by *OpenShortcut*. If the *ShortcutPath* property is null, *GetFullShortcutPath* calls *GetSpecialFolderPath*. Otherwise, it assigns the value of the *ShortcutPath* property to *Result*. It then adds the *ShortcutFileName* property to the end of the string. This is safe because *GetSpecialFolderPath* always returns a path that ends with a backslash, and the write method for the *ShortcutPath* property ensures that the property value always ends with a backslash. The write method for the *ShortcutFileName* property ensures that the filename always includes the .lnk extension.

GetSpecialFolderPath calls *SHGetSpecialFolderLocation* and passes the value of the *SpecialFolderLocation* property as the second parameter. This call loads the *ItemIdList* variable passed as the third parameter. Next, *GetSpecialFolderPath* calls *SHGetPathFromIdList*, passing two parameters. The first is the *ItemIdList* variable initialized by the call to *SHGetSpecialFolderLocation*, and the second is a char array, *CharStr*, into which the path will be placed. Finally, *CharStr* is assigned to the *Result* variable and a backslash is appended to the path.

The final step in the *OpenShortcut* method is the call to *GetPropertiesFromShortcut*. This method calls each of the get methods in the *IShellLink* interface and assigns the returned value to the corresponding property of *TWinShortcut*. For example, the first call is to the *IShellLink GetPath* method, which returns the path to the target file, i.e. the file to which the shortcut points. These calls are straightforward with two exceptions. If you create a shortcut manually in Windows, and the shortcut is to a program that requires command-line arguments, you type them in the Target edit box following the path to the EXE file. However, the command-line arguments are stored separately in the shortcut file and are retrieved with a separate call, *GetArguments*.

The call to *GetHotkey* returns the hotkey information in a single parameter of type Word. The virtual key code is stored in the low byte, and the modifier flags that indicate which shift keys were pressed are stored in the high-order byte. If you want to display the hotkey as text, or give users the ability to enter a hotkey, the easy way is to use the *THotkey* component from the Win32 page of the Component palette. The problem is that the *THotkey* component stores the virtual key code in its *HotKey* property, and the modifier flags in its *Modifiers* property. To make things worse, the values used to represent the Ctrl, Alt, Shift, and extended keys in the high byte of the value returned by *GetHotkey* aren't the same as the values used to represent the same keys in the *Modifiers* property of *THotkey*.

(Note: The extended-key flag indicates whether the keystroke message originated from one of the additional keys on the enhanced

keyboard. The extended keys consist of the Alt and Ctrl keys on the right-hand side of the keyboard; the Ins, Del, Home, End, Page Up, PageDown, and the arrow keys to the left of the numeric keypad; the NumLock key; the Break key; the PrtScr key; and the divide (/) and Enter keys in the numeric keypad. The extended-key flag is set if the key is an extended key.)

To make life easier for anyone using *TWinShortcut*, it has two properties, *Hotkey* and *HotkeyModifiers*, that are assignment-compatible with the properties of *THotkey*. The code following the call to *GetHotkey* converts the modifier flags from the form used by *GetHotkey* to the form used by the *HotkeyModifiers* property and by the *THotkey* component. The modifier constants used with *GetHotkey* and *SetHotkey* (HOTKEYF_ALT, HOTKEYF_CONTROL, HOTKEYF_SHIFT and HOTKEYF_EXT) are declared in the CommCtrl.pas unit. The constants used with the *THotkey* component's *Modifiers* property (*hkAlt*, *hkCtrl*, *hkShift*, and *hkExt*) are declared in the ComCtrls.pas unit.

Creating or saving a modified shortcut is handled by the *TWinShortcut*'s public *SaveShortcut* method. *SaveShortcut* begins by calling *PutPropertiesToShortcut*. This method calls the *IShellLink* put method for each property to assign the current value of the *TWinShortcut* properties to the corresponding shortcut properties. The only part of this process that is complex is converting the *HotkeyModifiers* property to the form required by the *SetHotkey* method. The series of **if** statements set the appropriate bits in the byte variable *HotKeyMods*. *SetHotkey* is called with a single-word parameter that's constructed by shifting *HotKeyMods* left eight bits to place it in the high-order byte of the word and adding the value of the *HotKey* property. Next, a call to *GetFullShortcutPath* returns the path to the link file. Finally, the *IPersistFile Save* method is called with the full path to the link file as a parameter. Again, the path must be cast first to a WideString, and then to a PWideChar.

Conclusion

You can create and modify any Windows shortcut using the methods of the *IShellLink* and *IPersistFile* interfaces implemented by the *ShellLink* object. Although this article doesn't cover every method in detail, it should give you everything you need for most shortcut operations. For more detailed information about the interfaces or any of their methods, consult the Win32 Programmers Reference online Help file that's installed with Delphi 5. ∆

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\OCT\DI200010BT.

Bill Todd is president of The Database Group, Inc., a database consulting and development firm based near Phoenix. He is a Contributing Editor to *Delphi Informant Magazine*, co-author of four database programming books, author of over 60 articles, and a member of Team Borland, providing technical support on the Borland Internet newsgroups. He is a frequent speaker at Borland Developer Conferences in the US and Europe. Bill is also a nationally known trainer and has taught Paradox and Delphi programming classes across the country and overseas. He was an instructor on the 1995, 1996, and 1997 Borland/Softbite Delphi World Tours. He can be reached at bill@dbginc.com.

Begin Listing One — *TWinShortcut*

```
unit WinShortcut;

interface

uses
  Windows, Messages, SysUtils, Classes, Graphics, Controls,
  Forms, Dialogs, ComObj, ShlObj, ShellAPI, ActiveX, Menus,
  ComCtrls;

type
  TWinShortcut = class(TComponent)
  private
    { Private declarations. }
    FArguments:           string;
    FDescription:         string;
    FHotkey:              Word;
    FHotKeyModifiers:     THKModifiers;
    FIconFile:            string;
    FIconIndex:           Integer;
    FShortcutFileName:    string;
    FShortcutPath:        string;
    FRunWindow:           Integer;
    FSpecialFolder:       Integer;
    FTarget:              string;
    FWorkingDirectory:    string;
  protected
    { Protected declarations. }
    FPersistFile: IPersistFile;
    FShellLink: IShellLink;
    function GetFullShortcutPath: string;
    procedure GetPropertiesFromShortcut;
    function GetSpecialFolderPath: string;
    procedure PutPropertiesToShortcut;
    procedure SetShortcutFileName(Value: string);
    procedure SetShortcutPath(Value: string);
    procedure SetSpecialFolder(Value: Integer);
  public
    { Public declarations. }
    constructor Create(AOwner: TComponent); override;
    destructor Destroy; override;
    procedure OpenShortcut;
    procedure SaveShortcut;
  published
    { Published declarations. }
    property Arguments: string
      read FArguments write FArguments;
    property Description: string
      read FDescription write FDescription;
    property HotKey: Word read FHotkey write FHotkey;
    property HotKeyModifiers: THKModifiers
      read FHotKeyModifiers write FHotKeyModifiers;
    property IconFile: string
      read FIconFile write FIconFile;
    property IconIndex: Integer
      read FIconIndex write FIconIndex;
    property RunWindow: Integer
      read FRunWindow write FRunWindow;
    property Target: string read FTarget write FTarget;
    property ShortcutFileName: string
      read FShortcutFileName write SetShortcutFileName;
    property ShortcutPath: string
      read FShortcutPath write SetShortcutPath;
    property SpecialFolder: Integer
      read FSpecialFolder write SetSpecialFolder;
    property WorkingDirectory: string
      read FWorkingDirectory write FWorkingDirectory;
  end;

procedure Register;

implementation

uses CommCtrl;
```

```
const
  Backslash = '\';
  LinkExtension = '.LNK';

{ ********* Constructor and Destructor ********** }

constructor TWinShortcut.Create(AOwner: TComponent);
begin
  { Create the ShellLink object and get an IShellLink and
    an IPersistFile reference to it. }
  inherited;
  FShellLink :=
    CreateComObject(CLSID_ShellLink) as IShellLink;
  FPersistFile := FShellLink as IPersistFile;
end;

destructor TWinShortcut.Destroy;
begin
  { Free the ShellLink object. }
  FShellLink := nil;
  FPersistFile := nil;
  inherited;
end;

{ ********* Property Getter and Setter Methods ********** }

procedure TWinShortcut.SetShortcutFileName(Value: string);
begin
  FShortcutFileName := Value;
  { If the file name does not end with the .LNK extension,
    add the extension. }
  if CompareText(ExtractFileExt(FShortcutFileName),
                 LinkExtension) <> 0 then
    FShortcutFileName := FShortcutFileName + LinkExtension;
end;

procedure TWinShortcut.SetShortcutPath(Value: string);
begin
  FShortcutPath := Value;
  { Make sure the path ends with a backslash. }
  if Copy(FShortcutPath,
     Length(FShortcutPath), 1) <> Backslash then
    FShortcutPath := FShortcutPath + Backslash;
end;

procedure TWinShortcut.SetSpecialFolder(Value: Integer);
begin
  FSpecialFolder := Value;
  { Clear the ShortcutPath when a value is assigned to the
    SpecialFolder property. The SpecialFolder property will
    not be used to get the path to the link file if the
    ShortcutPath property is not null. }
  FShortcutPath := '';
end;

{ ********* Custom Methods ********** }

function TWinShortcut.GetFullShortcutPath: string;
{ Gets the path to the shortcut file. If the ShortcutPath
  property is null, the path comes from the SpecialFolder
  property. }
begin
  if FShortcutPath = '' then
    Result := GetSpecialFolderPath
  else
    Result := FShortcutPath;
  Result := Result + FShortcutFileName;
end;

procedure TWinShortcut.GetPropertiesFromShortcut;
{ Calls the appropriate IShellLink method to get the value
  of each property of the link and assign that value to the
  corresponding property of this component. }
var
  CharStr:       array[0..MAX_PATH] of Char;
  WinFindData:   TWin32FindData;
  RunWin:        Integer;
  HotKeyWord:    Word;
  HotKeyMod:     Byte;
begin
  OleCheck(FShellLink.GetPath(CharStr, MAX_PATH,
           WinFindData, SLGP_UNCPRIORITY));
  Target := CharStr;
  OleCheck(FShellLink.GetArguments(CharStr, MAX_PATH));
  Arguments := CharStr;
  OleCheck(FShellLink.GetDescription(CharStr, MAX_PATH));
  Description := CharStr;
  OleCheck(
    FShellLink.GetWorkingDirectory(CharStr, MAX_PATH));
  WorkingDirectory := CharStr;
  OleCheck(FShellLink.GetIconLocation(CharStr, MAX_PATH,
                                      FIconIndex));
  IconFile := CharStr;
  OleCheck(FShellLink.GetShowCmd(RunWin));
  RunWindow := RunWin;
  OleCheck(FShellLink.GetHotkey(HotKeyWord));
  { Extract the HotKey and Modifier properties. }
  HotKey := HotKeyWord;
  HotKeyMod := Hi(HotKeyWord);
  if (HotKeyMod and HOTKEYF_ALT) = HOTKEYF_ALT then
    Include(FHotKeyModifiers, hkAlt);
  if (HotKeyMod and HOTKEYF_CONTROL) = HOTKEYF_CONTROL then
    Include(FHotKeyModifiers, hkCtrl);
  if (HotKeyMod and HOTKEYF_SHIFT) = HOTKEYF_SHIFT then
    Include(FHotKeyModifiers, hkShift);
  if (HotKeyMod and HOTKEYF_EXT) = HOTKEYF_EXT then
    Include(FHotKeyModifiers, hkExt);
end;

function TWinShortcut.GetSpecialFolderPath: string;
{ Returns the full path to the special folder specified in
  the SpecialFolder property. A backslash is appended to
  the path. }
var
  ItemIdList: PItemIdList;
  CharStr:    array[0..MAX_PATH] of Char;
begin
  OleCheck(ShGetSpecialFolderLocation(0, FSpecialFolder,
                                      ItemIdList));
  if ShGetPathFromIdList(ItemIdList, CharStr) then begin
    Result := CharStr;
    Result := Result + Backslash;
  end; // if
end;

procedure TWinShortcut.OpenShortcut;
{ Opens the shortcut and loads its properties into the
  component properties. }
var
  FullPath: string;
begin
  FullPath := GetFullShortcutPath;
  OleCheck(FPersistFile.Load(PWideChar(WideString(
    FullPath)), STGM_READWRITE));
  GetPropertiesFromShortcut;
end;

procedure TWinShortcut.PutPropertiesToShortcut;
{ Calls the appropriate IShellLink method to assign the
  value of each of the components properties to the
  corresponding property of the shortcut. }
var
  HotKeyMods: Byte;
begin
  HotKeyMods := 0;
  OleCheck(FShellLink.SetPath(PChar(FTarget)));
  OleCheck(FShellLink.SetIconLocation(PChar(FIconFile),
                                      FIconIndex));
  OleCheck(FShellLink.SetDescription(PChar(FDescription)));
  OleCheck(FShellLink.SetWorkingDirectory(PChar(
    FWorkingDirectory)));
```

```
  OleCheck(FShellLink.SetArguments(PChar(FArguments)));
  OleCheck(FShellLink.SetShowCmd(FRunWindow));
  if hkShift in FHotKeyModifiers then
    HotKeyMods := HotKeyMods or HOTKEYF_SHIFT;
  if hkAlt in FHotKeyModifiers then
    HotKeyMods := HotKeyMods or HOTKEYF_ALT;
  if hkCtrl in FHotKeyModifiers then
    HotKeyMods := HotKeyMods or HOTKEYF_CONTROL;
  if hkExt in FHotKeyModifiers then
    HotKeyMods := HotKeyMods or HOTKEYF_EXT;
  OleCheck(
    FShellLink.SetHotkey((HotKeyMods shl 8) + HotKey));
end;

procedure TWinShortcut.SaveShortcut;
{ Copies the component properties to the shortcut
  and saves it. }
var
  FullPath: string;
begin
  PutPropertiesToShortcut;
  FullPath := GetFullShortcutPath;
  OleCheck(FPersistFile.Save(PWideChar(
    WideString(FullPath)), True));
end;

procedure Register;
begin
  RegisterComponents('DI', [TWinShortcut]);
end;

end.
```

End Listing One

By Alex Fedorov and Natalia Elmanova

A Practical Guide to ADO Extensions

Part I: Using ADOX and JRO

Delphi is well known for its ability to access various databases. Until recently, however, this access has required the Borland Database Engine (BDE). As you probably know, Delphi 5 features an alternative technology called ADO Express. This set of components provides the object-oriented, high-level interface to Microsoft ActiveX Data Objects (ADO), which is a part of the Microsoft Universal Data Access architecture.

Although ADO and ADO Express have already been covered in *Delphi Informant Magazine*, several extensions to this data access technology deserve separate attention. This month, in Part I, we'll look at ADO Extensions for DDL and Security (ADOX), and the Jet and Replication Objects library (JRO). Next month, we'll explore ADO Multi-dimensional (ADO MD). [Visit http://msdn.microsoft.com/library/officedev/odeopg/deovradocomponentlibraries.htm for a full description of the ADO component libraries.]

Introduction to ADOX

ADOX can be used to perform various tasks not available using ADO alone. For example, you can extract information about users and/or create new user accounts. ADOX extends the ADO object model with 10 objects that can be used separately or in conjunction with ADO. You can use the ADO *Connection* object, for instance, to connect to a data source and extract metadata.

Metadata describes the database itself (e.g. tables, columns, indexes, keys, stored procedures, etc.), rather than the data it contains. SQL is used to define the metadata in most modern databases. Before ADOX, the only way to extract metadata from data sources using ADO was to use the *OpenSchema* method of the ADO *Connection* object. To create new database objects, you used the Data Definition Language (DDL) component of SQL and the ADO *Command* object. In other words, you were passing in SQL statements, which — obviously — necessitated a knowledge of SQL.

ADOX provides a way to manipulate metadata that doesn't require an understanding of SQL. Note that ADOX doesn't work with all databases in the world; its functionality is limited to Microsoft Access, Microsoft SQL Server, and a few databases from other vendors. For more information, visit http://www.microsoft.com/data.

The ADOX object model is shown in Figure 1, and many of its objects are briefly described in Figure 2. The top-level object in the ADOX object model is *Catalog*. It contains the *Tables*, *Views*, *Procedures*, *Users*, and *Groups* collections. The *Catalog* object can be used to open an existing database (through the ADO *Connection* object), or to create a new one. In

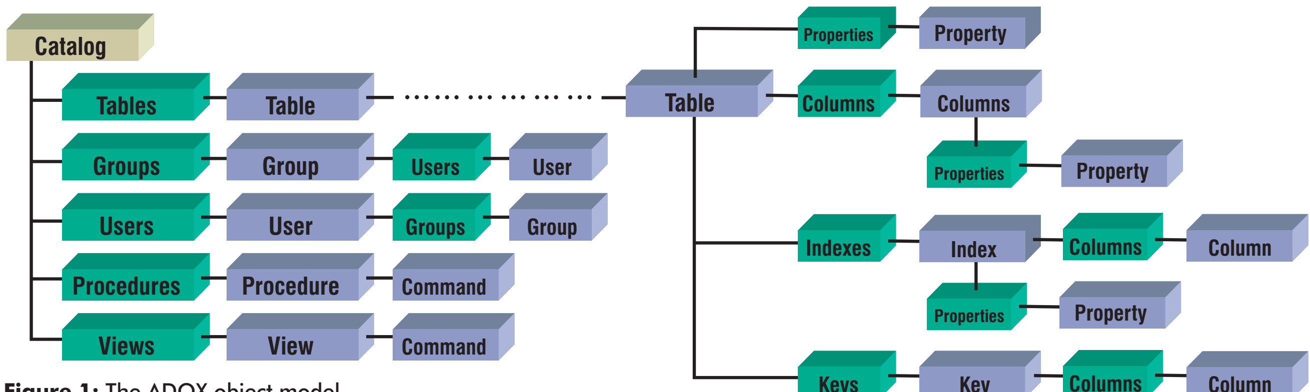


Figure 1: The ADOX object model.

Object	Description
Catalog	Represents the schema of the database, and provides access to collections of all the tables, procedures, users, and groups in a database.
Column	Represents a column in a table, or the columns involved in an index or key.
Connection	Used to provide a connection to a data source.
Group	Represents a group account that has access to a secured database.
Index	Represents an index on a table. Contains a collection of the *Column* objects upon which the index is based.
Key	Represents a key on a table. Contains a collection of the *Column* objects upon which the key is based.
Procedure	Represents a stored procedure or query.
Table	Represents a table in the database and provides access to columns, indexes, and keys.
User	Represents a user of a secured database.
View	Represents a view (a virtual table).

Figure 2: Selected ADOX objects.

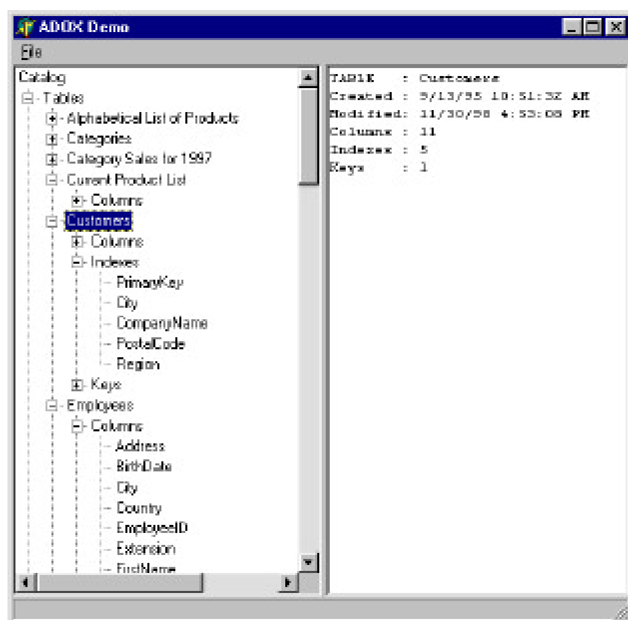


Figure 3: The demonstration application for this article loads the metadata of a selected database into a TreeView component.

the current version of ADO, we can only create new Jet 4.0 databases, but in future versions, this ability will be extended to other databases.

Once we have the *Catalog* object, we can work with *Tables*, *Procedures*, and *Views*. For example, by iterating the *Tables* collection, we can find what tables are in the database. Going deeper, we can check a *Table* object's *Columns*, *Indexes*, and *Keys* collections. By checking the properties of a database, we can get information about its metadata, and, for example, store it in a separate file or transfer it somewhere. Using the *Users* and *Groups* collections, we can obtain security information to find group accounts and users of a secured database. Note that this requires a secured database. In the case of Access, we must include the System.mdw database in the connection string to the data source.

The other, more exciting, thing about ADOX is that we can use it to create databases and objects from scratch. For example, we can create a Jet (Access) database, add tables, columns, indexes, and keys, and then programmatically or manually fill this newly created database with the information. This can be a great help in situations where we have raw data that needs to be organized in some way. In a general case, to create a new database, we use the *Catalog* object, and then use the *Add* method of the *Tables*, *Columns*, *Keys*, and *Indexes* collections to add the database objects to it.

Now that we've discussed the ADOX objects, let's use them to create a simple ADOX viewer.

Creating a Simple ADOX Viewer

Let's look at how to use ADOX objects in Delphi. We'll create an application that can:

- Show the database metadata in a tree view.
- Show properties of database objects.
- Show the source code of views and stored procedures.

To perform this task, we'll create a new project and place the following components onto the main form: MainMenu, TreeView, Memo, and StatusBar. The completed demonstration application is shown in Figure 3. (It's also available for download; see end of article for details.)

Next, we must include a reference to the ADOX type library, i.e. the Msadox.dll file. To do this, select **Project | Import Type Library** from the main menu of the Delphi IDE, then select **Microsoft ADO Ext. 2.1 for DDL and Security** from the list of available type libraries. To avoid conflicts with previously declared classes, rename the ADOX classes (e.g. *TTable* to *TTablexxx*), uncheck the **Generate Component Wrapper** checkbox (we only need the .pas file), and press the **Create Unit** button. This will create an ADOX_TLB.pas file — the interface unit to the ADOX type library. We'll need to refer to it in the **uses** clause of our main project unit. We'll also need to include the ADODB unit in the **uses** clause. Now we're ready to write code that uses ADOX objects. We'll create the **File | Open Catalog** menu item, and modify its *OnClick* event handler to appear as shown here:

```
procedure TForm1.OpenCatalog1Click(Sender: TObject);
begin
  // Get DataSourceName through standard MS dialog box.
  DS := PromptDataSource(Application.Handle, '');
  // If user selected one...
  if DS <> '' then
    BrowseData(DS);
end;
```

Here we use the *PromptDataSource* method, implemented in the ADODB unit, to display the standard Data Link Properties dialog box (see Figure 4). If you have the Microsoft Access sample databases installed, you may want to test with the Northwind.mdb database using the Microsoft Jet OLE DB provider.

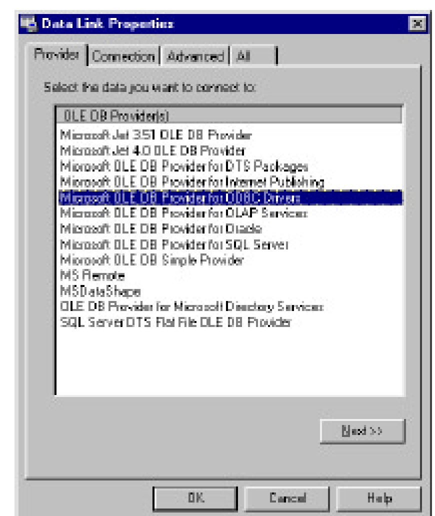


Figure 4: The Data Link Properties dialog box.

Once a data source is selected, the *BrowseData* procedure is called. The purpose of this procedure is to fill the TreeView component with metadata extracted from the data source. Figure 5 shows the code that does the job.

There are three loops that iterate through the *Tables*, *Views*, and *Procedures* collections of the *Catalog* object. Each object found is placed on the appropriate branch of the TreeView component. The *Tables* collection contains one or more *Table* objects, each of which should be processed to find *Columns*, *Indexes*, and *Keys* within it. This is done with the *ProceedTables* procedure, shown in Figure 6.

Again, we have three loops. In this procedure they iterate the *Columns*, *Indexes*, and *Keys* collections of the *Table* object.

Returning to the *BrowseData* procedure, we find that before the loop through the *Views* collection, we perform the following check:

```
if CheckViews(Catalog) then ...
```

```
procedure TForm1.BrowseData(DataSource: string);
var
  RootNode  : TTreeNode;
  OneNode   : TTreeNode;
  SubNode   : TTreeNode;
  I         : Integer;
  OldCursor : TCursor;
begin
  // Change the default cursor to an hourglass.
  OldCursor := Screen.Cursor;
  Screen.Cursor := crHourglass;
  StatusBar1.Panels[0].Text :=
    'Extracting metadata, please wait.';
  // Clear TreeView and Memo.
  ClearTree;
  Memo1.Lines.Clear;
  Application.ProcessMessages;
  // Connect to the data source.
  Catalog._Set_ActiveConnection(DataSource);
  RootNode := TreeView1.Items.Add(nil, 'Catalog');

  // Add Tables.
  OneNode  := TreeView1.Items.AddChild(RootNode, 'Tables');
  for I := 0 to Catalog.Tables.Count-1 do begin
    SubNode := TreeView1.Items.AddChild(
                 OneNode, Catalog.Tables[I].Name);
    // Process Columns, Indexes, and Keys.
    ProceedTables(Catalog.Tables[I], SubNode);
  end;
  // Add Views.
  if CheckViews(Catalog) then begin
    OneNode := TreeView1.Items.AddChild(RootNode, 'Views');
    for I := 0 to Catalog.Views.Count-1 do
      SubNode := TreeView1.Items.AddChild(
                   OneNode, Catalog.Views[I].Name);
  end;
  // Add Procedures.
  OneNode  := TreeView1.Items.AddChild(RootNode,
                                       'Procedures');
  for I := 0 to Catalog.Procedures.Count-1 do
    SubNode := TreeView1.Items.AddChild(
                 OneNode, Catalog.Procedures[I].Name);
  RootNode.Expand(False);

  // Restore the default cursor and clear the status bar.
  Screen.Cursor := OldCursor;
  StatusBar1.Panels[0].Text := '';
end;
```

Figure 5: Filling the TreeView component with metadata extracted from the data source.

This is done to avoid possible errors with data sources for which ADOX does not support *Views* collections. The *CheckViews* function is shown in Figure 7.

Now we have the TreeView component filled with metadata information. To get more information about this particular object, we need to implement the *OnChange* event handler for the TreeView component (see Figure 8).

As you can see, this is straightforward code that calls one of six procedures, depending on which node was clicked on the TreeView component. For example, the *ViewTables* procedure displays the number of objects within the table selected, and *ViewColumns*, *ViewIndexes*, and *ViewKeys* are used to study the properties of the *Column*, *Index*, and *Key* objects (see Figures 9, 10, and 11 respectively).

```
procedure TForm1.ProceedTables(T: Table; N: TTreeNode);
var
  I       : Integer;
  SubNode : TTreeNode;
begin
  // Add Columns.
  if T.Columns.Count > 0 then
    SubNode := TreeView1.Items.AddChild(N, 'Columns');
  for I := 0 to T.Columns.Count-1 do
    TreeView1.Items.AddChild(SubNode,
                             T.Columns.Item[I].Name);
  // Add Indexes.
  if T.Indexes.Count > 0 then
    SubNode := TreeView1.Items.AddChild(N, 'Indexes');
  for I := 0 to T.Indexes.Count-1 do
    TreeView1.Items.AddChild(SubNode,
                             T.Indexes.Item[I].Name);
  // Add Keys.
  if T.Keys.Count > 0 then
    SubNode := TreeView1.Items.AddChild(N, 'Keys');
  for I := 0 to T.Keys.Count-1 do
    TreeView1.Items.AddChild(SubNode, T.Keys.Item[I].Name);
end;
```

Figure 6: Processing a table's objects.

```
function CheckViews(C: _Catalog): Boolean;
var
  I : Integer;
begin
  try
    I := C.Views.Count;
    CheckViews := True;
  except
    CheckViews := False;
  end;
end;
```

Figure 7: The *CheckViews* function.

```
procedure TForm1.TreeView1Change(Sender: TObject;
  Node: TTreeNode);
begin
  if Node.Parent.Parent <> nil then
    case Node.Parent.Text[1] of
      'C' : ViewColumns(Node.Parent.Parent.Text,Node.Text);
      'I' : ViewIndexes(Node.Parent.Parent.Text,Node.Text);
      'K' : ViewKeys(Node.Parent.Parent.Text,Node.Text);
      'T' : ViewTables(Node.Text);
      'V' : ViewProps(Node.Text);
      'P' : ProcProps(Node.Text);
    end;
end;
```

Figure 8: The *OnChange* event handler for the TreeView component.

The *ViewProps* and *ProcProps* procedures display the source code of the view or stored procedure. The *ProcProps* procedure is shown in Figure 12. The *ViewProps* procedure is similar, so it's not shown here.

Here we use the fact that the stored procedure stored in the *Procedures* collection actually points to the ADO *Command* object. Thus, we use the *Get_Command* method to extract the *IDispatch* interface to the *Command* object, and use its *Get_CommandText* method to obtain the source code of the stored procedure.

Now we know how to use the ADOX object to retrieve metadata from various data sources, as well as display the resulting information. Another ADOX possibility, which we'll cover next, is the ability to create databases from scratch, without complex SQL DDL statements.

Property	Description
Attributes	Contains characteristics of a column.
DefinedSize	Contains the maximum size of a column.
NumericScale	Contains the scale of a numeric column.
ParentCatalog	Indicates the catalog to which this column belongs.
Precision	Contains the maximum precision of data in the column.
RelatedColumn	Contains the name of the related column for key columns.
SortOrder	Indicates the sorting order for a column.
Type	Contains the data type for the column values.

Figure 9: *Column* object properties.

Property	Description
Clustered	Indicates whether the index is clustered.
IndexNulls	Shows how null indexes are processed.
PrimaryKey	Indicates whether the index is the primary key.
Unique	Indicates whether the keys in the index must be unique.

Figure 10: *Index* object properties.

Property	Description
DeleteRule	Shows how primary key deletion is processed.
RelatedTable	Indicates the name of the foreign table for the foreign key.
Type	Contains the type of key.
UpdateRule	Shows how the primary key update is processed.

Figure 11: *Key* object properties.

```
procedure TForm1.ProcProps(Name: string);
var
  S       : string;
  Disp    : IDispatch;
  Command : _Command;
begin
  S := 'PROCEDURE : ' + Catalog.Procedures.Item[Name].Name;
  S := S + ^M^J + 'Created   : ' +
  VarToStr(Catalog.Procedures.Item[Name].DateCreated);
  S := S + ^M^J + 'Modified  : ' +
  VarToStr(Catalog.Procedures.Item[Name].DateModified);
  if CmdSupported(Catalog.Procedures.Item[Name]) then begin
    Disp := Catalog.Procedures.Item[Name].Get_Command;
    Command := Disp AS Command;
    S := S + ^M^J^M^J + Command.Get_CommandText;
  end;
  Memo1.Text := S;
end;
```

Figure 12: The *ProcProps* procedure displays a stored procedure's source code.

Creating Databases and Objects

The first step in creating a new database is to create a new instance of the *Catalog* object. This allows us to specify not only the type of the database to be created (through the OLE DB Provider), but also the location of the database file. Figure 13 shows how this can be done for an Access database.

This creates a new database of the specified type, at the specified location. After that, we can append tables and columns to the database. Here are the steps:
1) Create a new instance of the *Table* object.
2) Create a new instance of the *Column* object.
3) Specify the properties of the new column.
4) Add the *Column* object to the *Columns* collection of the *Table* object.

```
const
  BaseName = 'c:\data\demo.mdb';
  DS = 'Provider=Microsoft.Jet.OLEDB.4.0;Data Source=' +
       BaseName;
var
  Catalog : TADOXCatalog;
  ...
  // Create an instance of an ADOX Catalog object.
  Catalog := CoCatalog.Create;
  // If the database exists, delete it.
  if FileExists(BaseName) then
    DeleteFile(BaseName);
  // Create new .mdb file.
  Catalog.Create(DS);
  // Specify the active connection.
  Catalog._Set_ActiveConnection(DS);
  ...
```

Figure 13: Creating an Access database programmatically.

```
// STEP 1
// Create a new instance of the Table object.
Table := CoTable.Create;
// Give it a name...
 Table.Name := 'Customers';
// ...and specify the Catalog it belongs to.
 Table.ParentCatalog := Catalog;
// STEP 2
// Create a new instance of the Column object.
Column := CoColumn.Create;
with Column do begin
  ParentCatalog := Catalog;
  // STEP 3
  // Set the properties.
  Name  := 'CustID';
  Type_ := adInteger;
  Properties['Autoincrement'].Value := True;
  Properties['Description'].Value   := 'Customer ID';
end;
// STEP 4
// Append the Column to the table's Columns collection.
Table.Columns.Append(Column, 0, 0);
Column := nil;
// STEP 5
// Ceate more Columns and append them to the Table.
with Table.Columns do begin
  Append('FirstName', adVarWChar, 64);
  Append('LastName', adVarWChar, 64);
  Append('Phone', adVarWChar, 64);
  Append('Notes', adLongVarWChar, 128);
end;
// STEP 6
// Add the Table object to Tables collection of the
// Catalog object.
Catalog.Tables.Append(Table);
Catalog := nil;
```

Figure 14: Adding tables to a database.

5) Repeat steps 3 and 4 for each new column.
6) Add the *Table* object into *Tables* collection of the *Catalog* object.

The example in Figure 14 shows how these steps can be implemented.

After our table is created and its columns are defined, we can add indexes and keys as necessary. The following code shows how to create an index on the LastName column:

```
Index := CoIndex.Create;
with Index do begin
  Name := 'LastNameIndex';
  IndexNulls := adIndexNullsDisallow;
  Columns.Append('LastName', adVarWChar, 64);
  Columns['LastName'].SortOrder := adSortAscending;
end;
Table.Indexes.Append(Index, EmptyParam);
```

The logic of this code is straightforward. First, we create an instance of the *Index* object. Then we set its *Name* property, specify how null indexes are processed, associate it with the column, and — finally — add it to the *Indexes* collection of the *Table*. The same logic is used for keys.

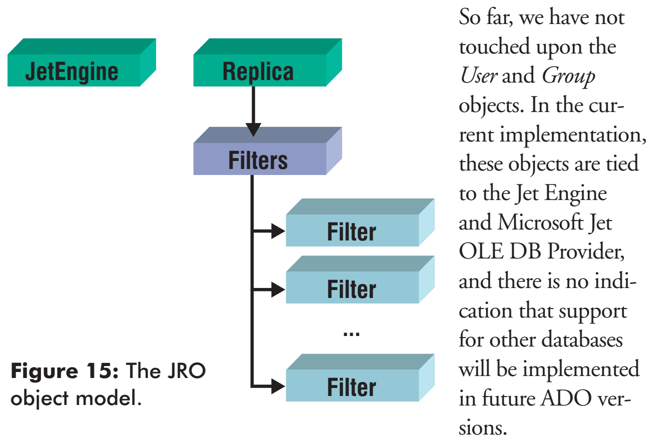


Figure 15: The JRO object model.

So far, we have not touched upon the *User* and *Group* objects. In the current implementation, these objects are tied to the Jet Engine and Microsoft Jet OLE DB Provider, and there is no indication that support for other databases will be implemented in future ADO versions.

```
const
  Provider = 'Provider=Microsoft.Jet.OLEDB.4.0;';
  // Replace these paths with the location of the
  // Microsoft Access sample databases.
  SrcMDB   = 'c:\data\northwind.mdb';
  DstMDB   = 'd:\data\newnorth.mdb';

procedure TForm1.Button1Click(Sender: TObject);
var
  JetEng : JetEngine;
  Src    : WideString;
  Dest   : WideString;
begin
  // Create an instance of the JetEngine object.
  JetEng := CoJetEngine.Create;
  // Specify the source.
  Src    := Provider + 'Data Source=' + SrcMDB;
  // And destination.
  Dest   := Provider + 'Data Source=' + DstMDB;

  // Check if the destination file exists and delete it.
  if FileExists(DstMDB) then
    DeleteFile(DstMDB);
  // Compact the database.
  JetEng.CompactDatabase(Src, Dest);
  // Free the JetEngine object instance.
  JetEng := nil;
end;
```

Figure 16: Creating a new, compacted copy of the database.

Using Jet and Replication Objects

The second ADO extension we'll cover this month is Jet and Replication Objects (JRO). Although the ADOX and ADO MD (which we'll cover next month) are able to work with various data sources, the JRO objects were implemented specifically to facilitate operations with Jet databases. In other words, contrary to ADOX and ADO MD, JRO objects can be used only with Access databases.

Introduction to JRO

Like other ADO extensions, JRO exposes the object model that contains objects, methods, and properties that can be used to create, modify, and synchronize replicas. The main object in the JRO object model is *Replica*, which can be used to create new replicas, check the properties of existing replicas, and synchronize changes with other replicas.

The JRO object model includes the *JetEngine* object, which exposes some features of the Jet engine. In particular, the *JetEngine* object can be used to compact the database, set password and encryption on databases, and refresh data from the memory cache. These objects form the hierarchy shown in Figure 15.

The topic of Jet replication *per se* is outside the scope of this article, so we'll just briefly describe the methods used for replication.

The first step in replication is to create a design master; indicate the database that will serve as a source for the replicas, and make that database replicable. This involves the *Replica* object and its *MakeReplicable* method. Then we need to change the replicability status of the database objects; the *GetObjectReplicability* and *SetObjectReplicability* methods of the *Replica* object are used for this. After that, depending on the task, we can create either the partial or full replica of the objects that are replicable in the design master.

Next, we can define some update rules. The *Filter* object is used for this purpose. Finally, we can synchronize data in two replicas. There can be direct or indirect synchronization or synchronization over the Internet. In the latter case, we need to use the Replication Manager that comes with Microsoft Office Developer.

To use the JRO library in your Delphi applications, use the Import Type Library dialog box to select the Microsoft Jet and Replication Objects 2.1 Library (Version 2.1) and press the Install button. This will create the JRO_TLB unit, which must then be included in your code to access the objects exposed by the JRO library.

Using the *JetEngine* Object

The *JetEngine* object can be used to compact the database and refresh data from cache. Figure 16 shows how to compact the Northwind.mdb database, then create a new compacted copy named Newnorth.mdb.

Without going into the inner depths of the Microsoft Jet Engine, let's outline what really happens when we compact a database:

- Tables pages are reorganized. After being compacted, they reside in adjacent database pages. This gives us greater performance since the table is no longer fragmented.
- Unused space is reclaimed by the deletion of objects and records that are marked as deleted.
- AutoNumber fields are reset so the next value allocated will be in the continuous sequence from the highest current value.
- The table statistics used for query optimization are updated.
- Since the database statistics were changed, all queries are flagged so they will be recompiled the next time the query is executed.

Conclusion

This article introduces two ADO extensions: ADO Extensions for DDL and Security (ADOX), and Jet and Replication Objects (JRO). We've seen how to use ADOX objects to retrieve metadata from data sources, and how these objects can be used to create databases from scratch. We've also seen how the JRO can be used to compact Jet databases, and have briefly outlined the replication process.

Next month, we'll discuss ADO Multi-dimensional (ADO MD), which is used for access to multi-dimensional data sources. See you then. Δ

The files referenced in this article are available on the Delphi Informant Magazine Complete Works CD in INFORM\00\OCT\DI200010AF.

Alex Fedorov is an Executive Editor for *ComputerPress* magazine, published in Moscow. He was one of the co-authors of *Professional Active Server Pages 2.0* [Wrox Press, 1998] and *ASP 2.0 Programmer's Reference* [Wrox Press, 1999]. Natalia Elmanova, Ph.D., is an Associate Professor of the Sechenov's Moscow Medical Academy, and a freelance Delphi/C++Builder programmer, trainer, and consultant. She was a speaker at the 10th Annual Inprise/Borland Conference. Natalia and Alex are authors of *Advanced Delphi Developer's Guide to ADO* [Wordware Publishing, 2000], and several programming books written in Russian. You can visit their Web site at http://d5ado.homepage.com.

NEW & USED

By Robert Leahey

IBObjects 3.4

Harness the Power of InterBase

In *Delphi Informant Magazine's* most recent Readers Choice Awards (see the April 2000 issue), Jason Wharton's InterBase Objects (IBObjects) collected the award for Best Database Connectivity. So I decided to have a peek at IBObjects and find out why. I examined IBObjects version 3.4b for this review, and found it to be an impressive set of components designed to allow developers to harness the power of InterBase in their applications.

IBObjects has two data access paradigms: *TDataSet*, a connectivity solution based on descendants of Delphi's *TDataSet*, and Native, a custom connectivity solution for client/server applications.

The *TDataSet* paradigm provides a set of data access controls designed to make converting applications to IBObjects as painless as possible. *TIBODatabase*, *TIBOTable*, and *TIBOQuery* are designed to simply replace their corresponding controls (*TDatabase*, *TTable*, and *TQuery*). Because virtually all of the BDE functionality is supported, complete replacement can be achieved with minimal fuss.

The Native paradigm provides a set of components designed as a client/server solution that avoids the problems of *TDataSet*'s desktop-centric nature.

In addition to these two data access paradigms, IBObjects provides a suite of components intended to automate many of the tasks associated with writing database applications. There are over two dozen data-aware controls, some of which provide familiar functionality, such as IB_Edit and IB_DateTimePicker, and some of which are highly specialized, such as IB_Ledger and IB_IncSearch. The latter provides for incremental searches in the descendants of TIB_BDataSet, while the former is a robust control with a powerful property editor for creating data-aware ledgers. I can't tell you how much I wish I'd had that for my last accounting application.

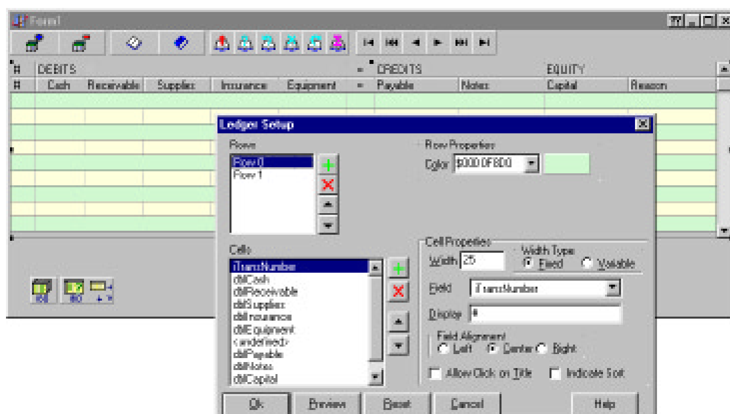


Figure 1: TIB_Ledger and Setup dialog box.

Figure 1 displays the TIB_Ledger control and its Cells property editor. Notice the amount of control and level of detail afforded by this dialog box. Not only is each cell assigned its own field and formatting, but the user can define multiple rows with differing cell layouts.

IBObjects features eight custom toolbars, aimed, again, at automating some standard functionality. There's a connection bar, a transaction bar, and a search bar. Each of these provide functionality in place by simply dropping them on a form and hooking them up to the appropriate datasource. No more writing the same old, mind-numbing search code for that button's event handler. Above the ledger control, in Figure 1, you'll see four of the custom toolbars.

In addition, there are nine dialog components that provide pre-built dialog boxes, such as lookups, exporting, browsing, and SQL monitoring.

Suffice it to say, IBObjects provides an impressive feature set: two-sided data connectivity, integrated data controls, toolbars, and dialog boxes. It's beyond the scope of this review to cover all of the functionality available, so let's take a closer look at two things: ease of transition and the test bench.

Making the Change

Obviously, one test for IBObjects is how easily a developer can convert an existing BDE application to use the IBObjects connectivity components. I decided to try converting one of our end-user sample applications that uses InterBase data. This project uses standard BDE controls (*TTable*, *TDatabase*, etc.) to access data for reporting, and to store user-created reports. The test was to replace all existing BDE components with IBObjects components, and to see what — if anything — would break.

I didn't relish the thought of replacing the controls and resetting properties by hand, so I opened the main form as text and simply replaced all instances of *TDatabase* with *TIBODatabase*, all *TTable*s with *TIBOTable*s, and all *TQuery*s with *TIBOQuery*s. When I viewed the form, to my great surprise, only two messages about missing properties popped up.

"That's promising," I thought, vowing not to get excited yet.

I did have to redirect my *TIBODatabase* at the proper data, but, buoyed by my early success, I decided to be daring and simply run the application, thinking my afternoon would be dedicated to tweaking settings and chasing bugs. Hey! The application was running!

I put it through its paces, creating reports, printing, saving, etc. No problems at all. I had converted our BDE + InterBase reporting application to an IBObjects + InterBase reporting application in five minutes and one property change.

Depending on the complexity of your application, you may need to do some preparation before trying to convert it using this technique. Jason's BDE-to-IBObjects Conversion Guide supplies an ever-shrinking list of properties and methods of the BDE controls that are currently unsupported by IBObjects. Browsing this list will help you decide how best to proceed when converting your application.

Connection	Average Completion Time
BDE	59.826 seconds
IBObjects	40.863 seconds

Figure 2: Results from the first test.

Connection	Average Completion Time
BDE	1 minute 53.036 seconds
IBObjects	1 minute 44.328 seconds

Figure 3: Results of the second test.

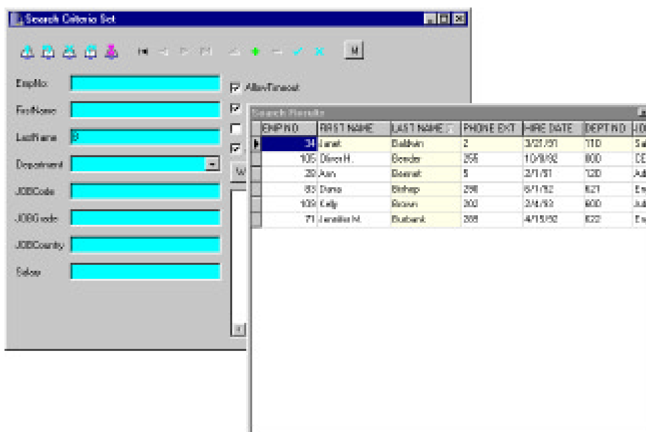


Figure 4: Automated searches.

The Test Bench

Another factor when considering a switch to IBObjects is, of course, speed. There's not much point in making the switch if the tool doesn't perform, right? Fear not. In my testing, IBObjects demonstrated a considerable edge in performance when compared to Delphi's BDE connectivity controls.

In my first test, I created a simple application, which queried and joined two InterBase tables, returning a result set of more than 123,000 records. Using Raize Software's CodeSite, I sent elapsed-time messages at various points in the test. I built two nearly identical versions of the application, one using *TDatabase* and *TQuery*, the other using *TIBODatabase* and *TIBOQuery*.

With a dozen test runs, the average times were as shown in Figure 2. Notice the significant edge IBObjects holds over the BDE.

For my second test, I again used a reporting scenario. I created an application that displayed a report based on that same 123,000+ record result set, and set the report component to make two passes. This would ensure a complete traversal of the data in one contiguous effort.

This time there was less disparity between the results, but IBObjects still gets the nod. For this 2,625-page report, the average results are shown in Figure 3.

The superior performance of IBObjects, combined with the ease with which I was able to convert my application to use its data access controls, makes a strong argument for its use. It's definitely worth your time to download the evaluation version of IBObjects.

Automation

While the advantages of IBObjects are many and varied, one of the most noteworthy is the amount of automation that it provides. Anyone who has written a client/server database application in Delphi can recount how much code they had to write for menial tasks, such as lookup dialog boxes, record counts, etc. Using the IBObjects controls in concert can automate much of this.

Take searching as an example. In Figure 4 you'll see two forms. This is a client/server application built with IBObjects. The application is currently in search mode (note that the IBObjects data controls have turned blue to indicate that they are accepting search criteria). The second form, with the grid, is displaying the result set of the search (records where the Last Name begins with "B"). This search dialog box was created with a small amount of code and just a few property settings. The IBObjects controls are working together to automate the search.

Granted, this is a simple example, but it illustrates the wealth of functionality built into these components. IBObjects greatly simplifies the act of writing client/server applications with InterBase by automating much of the task.

Other Thoughts

Wharton offers an intriguing license for IBObjects, called the Trustware License. Basically, the thought is that if you don't think you'll make money using it, you don't pay for the software. When, however, you begin to show a profit, Jason trusts you to pay for your license. The details of this generous license are on the IBObjects Web site at http://www.ibobjects.com/ibo_trustware.html.

For those who know InterBase, IBObjects is a must-have tool set. If you don't know InterBase, IBObjects is a good reason to learn it.

IBObjects connectivity tools provide a thin wrapper for the InterBase API. This is a very good thing if you already know InterBase; everything you're used to dealing with is available right there in Delphi's Object Inspector, or in IBObjects' property editors. This wealth of properties, however, can be daunting if you're not that familiar with InterBase. IBObjects doesn't candy-coat anything for you; you'll have to know or learn InterBase well to really understand what these components can do for you.

Obviously no software is perfect, and we should run screaming from reviews that try to tell us otherwise, but IBObjects' problems, at least as I have seen, have been merely a matter of falling afoul of my personal preferences.

Maybe I'm spoiled, or perhaps I've read a little too much Alan Cooper, but I have come to expect software packages to be well-behaved and to pamper me upon installation. I want to have a single executable file that I can run to install the software. When it's done, I want to launch Delphi and find my nicely installed new package (complete with Delphi-integrated help) waiting for me. This is rarely the case with third-party products, and it was not my experience with IBObjects. Installation was a chore, requiring me to move files, integrate the help, compile packages. Not a big deal, but, as I said, I'm spoiled and want to be taken care of.

Likewise, many IBObjects demos weren't set up to handle non-standard installation paths. I continually paid for my choice of having installed my InterBase sample data to my D drive. However, in each demo, once I updated the DB path, things went well. IBObjects exception messages are also a bit verbose and unclear, and the documentation isn't complete, although it does cover much of the product.

Having asked Jason Wharton about these issues, it seems he already plans to deal with them. He's building an installation routine that will drastically simplify the installation process. He's also working with a technical writer to produce more thorough documentation, and a "Getting Started" guide. Jason is obviously responsive, professional, and interested in improving his product. This reflects well on the future of IBObjects.

Informant Fact File

Jason Wharton's InterBase Objects offers a wide array of data-aware components with a high degree of interoperability, as well as two different sets of data connectivity controls. One set is for building high-performance client/server applications, and the other mirrors Delphi's *TDataSet* architecture, allowing developers to simply replace existing BDE controls with the IBObjects controls.

Jason Wharton
619 N. Macdonald St.
Mesa, AZ 85201

E-Mail: jwharton@ibobjects.com
Web Site: http://www.ibobjects.com
Price: Full-source license, US$395; full-source upgrade license, US$175; partial-source license, US$195; partial-source upgrade license, US$75; partial-source to full-source upgrade, US$215.

Conclusion

InterBase Objects is an outstanding product, and I strongly recommend it for those doing InterBase development. The improved performance attained by simply using the IBObjects connectivity controls is reason enough to warrant the switch, but when the wide range of time and effort saving IBObjects components is also considered, the true value of IBObjects becomes quite evident. Δ

Robert is Director of Product Management of ReportBuilder for Digital Metaphors Corp. He graduated with a degree in Music Theory from the University of North Texas, but has been writing code since his Apple II+ and AppleBasic days. He has been programming in Object Pascal since Delphi 1 and currently resides in Texas with his wife and daughters.

table of CONTENTS

Delphi Graphics and Game Programming Exposed! with DirectX

I was delighted when *Delphi Informant Magazine* asked if I would review John Ayres' new book, *Delphi Graphics and Game Programming Exposed! with DirectX*. As the title suggests, the book provides a foundation for graphics and game programming with DirectX — a subject in which I am very interested.

Quite appropriately, Ayres begins with an introduction to game programming as a specialization (including its ups and downs) and a detailed exposition of its essential elements. Although these two chapters will be must-reading for new game programmers, those with some experience in this domain may want to quickly skim them and delve into the remaining chapters, which expose the essential techniques. One of the high points of Chapter 2 is the case study in which Ayres builds a simple Space Invaders-type game to demonstrate the game loop and other principles discussed in this chapter.

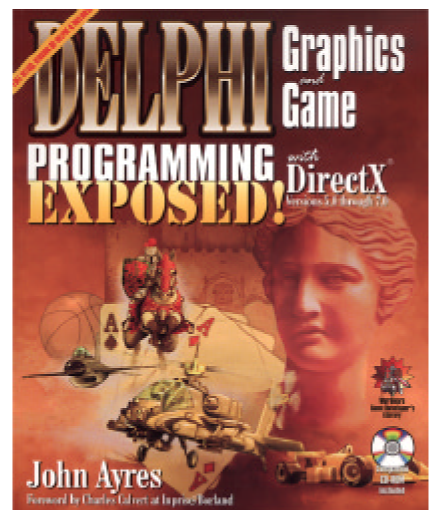
The majority of the remaining chapters use DirectX. To make this technology available, Ayres uses Erik Unger's Project JEDI DirectX header conversion as a foundation, and shows how to use the various units and utilities that make up that library. (Before you run any of the programs on the CD-ROM, be sure to include the two directories containing the JEDI files on the Delphi path.)

Before introducing the DirectX technologies, Ayres provides a basic introduction to graphics programming. He explains how to work with graphics elements such as pixels, bitmaps, and the *Canvas* object; he also discusses important concepts such as color components, color depth, and video modes. He presents techniques for drawing text, manipulating palettes, and calling GDI functions. As with most of the other chapters, there are many useful demonstration programs. Many chapters have 10 or more example projects.

Chapter 4 provides a fifty-page introduction to DirectX. It discusses the various components included in this essential game-writing technology, with emphasis on DirectDraw. The author explains how to use the various available functions to perform basic graphics operations, such as working with display modes, surfaces, and bitmaps. The next two chapters build on this foundation, exposing techniques for working with palettes and sprites. These techniques are presented in the context of game production — solving game development problems, such as collision detection in an action game.

DirectX also provides support for user input with the DirectInput component. Chapters 7 and 8 deal with this vital topic, with the latter chapter explaining a relatively new technique, Force Feedback. As in the previous chapters, Ayres begins by explaining theory and goes on to develop some actual examples in Delphi. He includes important notes, tips, and warnings — all highlighted for easy recognition. He includes important technical terms in two locations: where the terms appear in the text, and in a glossary at the end of the book. I strongly endorse this excellent approach.

There are other factors essential to the creation of a successful game: sound and music, special effects, and optimal performance.



The last several chapters discuss these issues, and more. The last chapter, "Putting It All Together," completes the sample application begun in Chapter 2. This is a fitting way to conclude this well-crafted, informative treatise. I strongly recommend this book for anyone who will be programming games in Delphi, who wants to add special graphical effects to their Delphi applications, or who wants to work with DirectX in Delphi.

— Alan C. Moore, Ph.D.

Delphi Graphics and Game Programming Exposed! with DirectX by John Ayres,
Wordware Publishing, Inc.,
2320 Los Rios Blvd., #200
Plano, TX 75074,
http://www.wordware.com.

ISBN: 1-55622-637-3
Price: US$59.95 (544 pages, CD-ROM)

Paint Your Editor

How much time do you spend debugging? Unless you're perfect, or you're not a programmer (is there a connection there?), you probably spend more time debugging than you care to admit.

If you're anything like me, what you like most about programming is designing and coding. Analysis must be done (or it's done by somebody else), but it's something to get out of the way so you can get to the interesting and creative work. Simply put, debugging is a necessary evil. What can be done to make this process less of a drudgery and more productive — so we can get through it and back to coding?

Something I find useful is modifying the appearance of the editor. By doing this, you acquire visual clues that enable you to see exactly what it is you are looking at in the editor. Using colors to differentiate strings from numbers, keywords from identifiers, etc., can be a great help in cutting through the haze of information overload we're bombarded with as we try to get to the root of the problem.

```
procedure PopulateRichMain;
var
  ltr: String;
begin
  Delete(MainPortion,Pos('~',MainPortion),1);
  with richMain.SelAttributes do begin
    Color := DEFAULT_COLOR;
    Style := [fsBold];
  end;
  if Pos('~',MainPortion) = 0 then begin
    richMain.SelText := MainPortion;
    Exit;
  end;
  while Pos('~',MainPortion)>0 do begin
    richMain.SelText := Copy(MainPortion,1,Pos('~',MainPortion)-1);
    if (Pos('~',MainPortion)) <> (Length(MainPortion)) then
      ltr := MainPortion[Pos('~',MainPortion)+1];
    with richMain.SelAttributes do begin
      Color := XREF_COLOR;
      Style := [];
    end;
    if (Pos('~',MainPortion)) <> (Length(MainPortion)) then
      richMain.SelText := ' '+ltr;
    with richMain.SelAttributes do begin
      Color := DEFAULT_COLOR;
      Style := [fsBold];
    end;
    Delete(MainPortion,1,Pos('~',MainPortion)+1);
  end;
end;
```

Figure 1: Use colors to modify the appearance of the editor.

It's been said that a picture is worth 1,024 words (or so). Figure 1 illustrates what I mean. As you can see, it's easy to identify the various elements of the code because they're color-coded (no pun intended!). Compare that to how your editor looks in its default configuration — black on white! Boring! Blah! Unimaginative! Unsophisticated! Unhelpful! I submit also that this is easier on the eyes. And we need all the help we can get to avoid carpal retinal syndrome, right?

Of course, if you don't care for my choice of colors, you can select your own to suit your fancy and sensibilities. I must admit, I love my color scheme and would rather fight than switch, but then again, a good portion of the blood flowing through my veins is "Port-a-gee," and we are not exactly known for subtlety in choice of color (it's not politically incorrect when you're poking fun at yourself, I strongly assert!).

So, do you want to take the plunge and spice up your debugging experience? It's easy. Here's what you need to do:

1) In Delphi 5, select Tools | Editor Options to display the Editor Properties dialog box; then select the Color tab as shown in Figure 2. For Delphi 4, select Tools | Environment Options instead.
2) Select an item in the Element list box at the left, then click on the color you want to set as the foreground color for the selected element. "FG" will appear in that color indicating your choice.
3) Right-click on the color you want to set as the background color for the selected element ("BG" will appear in that color). Unless you're crazy, you'll want to stick with one color for the background of each element. As probably did not escape your notice, I have chosen black. You will doubtless want to choose either a very light background, with dark foreground colors, or vice versa (as I have).
4) Repeat steps 2-5 for each element you want to set.

If you want to copy my color scheme (especially you Portuguese developers out there), check out the table in Figure 3.

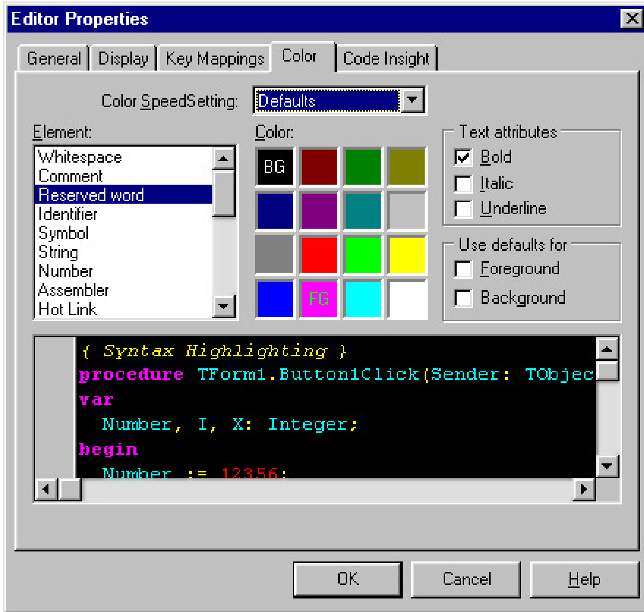


Figure 2: The Color page of the Editor Properties dialog box.

Element	Foreground	Background
Whitespace	Blue	Black
Comment	Yellow (italics)	Black
Reserved Word	Magenta	Black
Identifier	Aqua	Black
Symbol	Yellow	Black
String	Lime	Black
Number	Red	Black
Search Match	White	Black

Figure 3: My color scheme.

There are two certainties in life for programmers: debugging and taxes. You can ease the pain a little by colorizing that tired old Notepad-look-alike. Depending on your perspective and personal history, it may bring back memories of Turbo Pascal, or it may seem like a leap into the next era of RAD coding. At any rate, the improved comprehension afforded by highlighting the various elements of your (or somebody else's) code should augment the enjoyment of working with Delphi, and simultaneously reduce the amount of time you spend locating and eradicating those pesky bugs ("anomalies" to the intelligentsia; "issues" to the euphemism inclined/politically correct). Happy debugging! Δ

— *Clay Shannon*

Clay Shannon is an independent Delphi consultant based in northern Idaho. He is available for Delphi consulting work in the greater Spokane/Coeur d'Alene areas, remote development (no job too small!), and short-term or part-time assignments in other locales. Clay is a certified Delphi 5 developer, and is the author of *Developer's Guide to Delphi Troubleshooting* [Wordware, 1999]. You can reach him at BClayShannon@aol.com.

BorCon 2000 a.k.a. The Kylix Konvention

This was my fifth Borland Conference (hardly anyone is saying "Inprise" these days), and like the first one I attended, it took me to southern California — this time San Diego. The focus of this conference, as you might guess, was the important move to support Linux with versions of Delphi and C++Builder. Code-named Project Kylix, this development has engendered more excitement than anything Borland has initiated since its first release of Delphi.

All the other features that make Borland conferences so popular were present as well. The opening show — I mean keynote — was another spectacular multimedia event based on the popular movie, *The Matrix*, with company CEO Dale Fuller, David Intersimone, and others making entrances to its powerful theme music. The carefully crafted presentation was punctuated with some of the most powerful scenes from the movie, scenes that underlined aspects of the company's emerging philosophy.

That philosophy contains some elements that will please many who have been critical of it in the past. For one thing, the company has returned to its original developer-centered focus. It is also committed to achieving success through responsible action. This means that it will "not release any product before its time," very good news for all of us who remember early versions of Delphi 4.

The most important new element of this philosophy is "platform independence." This commitment goes beyond just support for Windows and the platforms supported by Linux. In fact, Fuller made it clear that the company was open to supporting new platforms that market demands justified. To punctuate some of these new directions, a Macintosh computer was unveiled and used briefly during the opening keynote. Delphi for the Mac? This may be stretching expectations a bit, but then who knows? JBuilder for the Mac is defiantly coming, and was demonstrated in a technology keynote by Blake Stone, one of JBuilder's architects. The key point to remember is this slogan: "The platform is the Net." Support for more traditional platforms — specific computers and/or operating systems — will be driven by demand. Good business sense, in my view.

As he did at last year's conference, Fuller took questions from the audience. One person asked when Inprise would have a profitable quarter. The CEO pointed out that the company was in better shape now than at any time in recent memory, that he would not take shortcuts (such as releasing a product prematurely) simply to increase profitability, and then shifted the burden back to the audience by encouraging everyone to become "Delphi Evangelists," becoming part of the new campaign to convert Visual Basic programmers to Delphi.

Ray Lischner asked about what he viewed as a "scattershot approach" where the company would release a foundation version of one product over here, and open-source another product over there. He wondered if there was a coherent strategy in place to map Borland's future development. I'm not certain that Fuller answered the question completely, but he did point out that the company could not open-source or release foundation versions of all its products, since it was, after all, a for-profit company. Still, Ray's questions will hopefully give Borland executives something to ponder as they consider important marketing decisions in the coming year. For example, if people could download a very basic version of Delphi with a few controls and no source, would they then be inspired to purchase it?

There are other marketing issues that weren't discussed in open sessions. As an academic, I very much want to see Delphi become more of a player in University Computer Science curricula. If Visual Basic can be used to teach beginning classes, why not Delphi? I promised some of the Borland people I spoke with that I will do what I can to help make this happen.

As in past years, this year's conference provided a surplus of sessions on Delphi and other tools. Again, I attended one of the pre-conference tutorials, a four-hour session in which Mark Miller presented "Design Patterns in Delphi." Mark did an outstanding job of presenting the Singleton, Composite, Factory, Proxy, Observer, Iterator, and Flyweight patterns. With each he discussed its applicability to a programming challenge, its advantages and disadvantages, and demonstrated its use in an actual Delphi application.

Mark's sessions are wild — exciting to many, exasperating to some. He loves to take questions from the audience, explore interesting tangents that come up, and interject wonderful humor and anecdotes. After editing his slides a few times in front of the audience and making other spontaneous changes, he shifted attention away from possible criticisms about his style by suggesting to the audience that when they attended a session by his friend Ray Konopka, they should evaluate the latter with, "He's too professional, too prepared." Personally I have no problem with Mark's approach, and go out of my way to attend his sessions.

Delphi 6. Of course, again this year the vast majority of participants at the conference were Delphi folk. For that reason there was a good deal of interest in the newest version of Delphi. I will be writing a column on Delphi 6 as soon as it ships, so I'm not going to go into great detail now. This year we'll see a number of new and expanded units. The Math unit will be expanded, with exciting new developments in the world of variants. There is a new StrUtils unit that will feature a plethora of string conversion routines. The R&D team has made enhancements to the *TCollection* and *TList* classes.

One of the most exciting new developments is the support for Web development. There will be a test Web server built into Delphi 6, and support for working with XML documents. Also, we saw the beginnings of a Delphi scripting language that can be used within HTML documents. The idea, of course, is to enable us to leverage our Delphi skills even more to develop Web applications.

Kylix. Many of the sessions I attended had to do with Kylix, the code-name for Delphi for Linux. Turbo Power's Gary Frerking provided an excellent introduction to Linux for newbies such as me. Charlie Calvert and David Intersimone presented an exciting introduction to Kylix. Let me make a prediction: The introduction of Delphi for Linux will be more monumental than the introduction of Delphi 1. Further, this implementation of Delphi will be the major focus of next year's conference, with a stampede of Linux developers eager to write GUI applications.

What specifications for Kylix did we learn about at the conference? Of course the Linux version of Delphi will be a component-based, visual development tool to rapidly produce native Linux applications. Its native code generator will use the ELF object format, allow two-way visual development (just as in Delphi for Windows), and be able to produce the three common types of applications: desktop, database, and Web.

Replacing Delphi for Windows' Visual Component Library (VCL), CLX (pronounced clicks) will provide a library of native Linux components. CLX comes in four flavors, with only the first one, VisualCLX, restricted to Linux. There will also be BaseCLX, providing basic components and system support; DataCLX, providing client data access; and NetCLX, providing support for Internet and related programming. To help ensure compatibility and ease of porting, there will be a new file version for Delphi for Linux form files (instead of .DFM). In his session entitled "New Language and RTL Features in Kylix," Danny Thorpe outlined additions, deletions, and changes to this exciting new flavor of Delphi.

Although much of the standard Delphi tool-set will be there, the stand-alone assembler (TASM) and resource compiler won't be included. There will be new conditional defines in both flavors of Delphi to support cross-platform development. There will be increased support for wide strings and variants (including custom variants). Of course, all Windows-specific units (including low-level multimedia support, sad to say) will be gone. Clearly, there will be further challenges for Project JEDI in the new Kylix world, and several JEDI knights are already working in that arena.

This was probably the most enjoyable conference so far. Personally, I had the gratification of having my new Delphi multimedia book released simultaneously with the beginning of the conference. I met many old and new friends, including a couple of folks involved with Project JEDI. Unfortunately, we were not able to schedule a JEDI Birds-of-a-Feather session this year. If you think I'm excited about Kylix and the future of Delphi, you're right. I can hardly wait for next year's conference. ∆

— *Alan C. Moore, Ph.D.*

Alan Moore is a Professor of Music at Kentucky State University, specializing in music composition and music theory. He has been developing education-related applications with the Borland languages for more than 10 years. He has published a number of articles in various technical journals. Using Delphi, he specializes in writing custom components and implementing multimedia capabilities in applications, particularly sound and music. You can reach Alan on the Internet at acmdoc@aol.com.

table of CONTENTS